The Rustonomicon

The Rust Team

2015-09-12

Contents

1 Introduction

The Dark Arts of Advanced and Unsafe Rust Programming

NOTE: This is a draft document, and may contain serious errors

> Instead of the programs I had hoped for, there came only a shuddering blackness and ineffable loneliness; and I saw at last a fearful truth which no one had ever dared to breathe before — the unwhisperable secret of secrets — The fact that this language of stone and stridor is not a sentient perpetuation of Rust as London is of Old London and Paris of Old Paris, but that it is in fact quite unsafe, its sprawling body imperfectly embalmed and infested with queer animate things which have nothing to do with it as it was in compilation.

This book digs into all the awful details that are necessary to understand in order to write correct Unsafe Rust programs. Due to the nature of this problem, it may lead to unleashing untold horrors that shatter your psyche into a billion infinitesimal fragments of despair.

Should you wish a long and happy career of writing Rust programs, you should turn back now and forget you ever saw this book. It is not necessary. However if you intend to write unsafe code – or just want to dig into the guts of the language – this book contains invaluable information.

Unlike The Book[1] we will be assuming considerable prior knowledge. In particular, you should be comfortable with basic systems programming and Rust. If you don't feel comfortable with these topics, you should consider reading The Book[2] first. Though we will not be assuming that you have, and will take care to occasionally give a refresher on the basics where appropriate. You can skip straight to this book if you want; just know that we won't be explaining everything from the ground up.

To be clear, this book goes into deep detail. We're going to dig into exception-safety, pointer aliasing, memory models, and even some type-theory. We will also be spending a lot of time talking about the different kinds of safety and guarantees.

[1] http://doc.rust-lang.org/book/
[2] http://doc.rust-lang.org/book/

2

Meet Safe and Unsafe

Programmers in safe "high-level" languages face a fundamental dilemma. On one hand, it would be *really* great to just say what you want and not worry about how it's done. On the other hand, that can lead to unacceptably poor performance. It may be necessary to drop down to less clear or idiomatic practices to get the performance characteristics you want. Or maybe you just throw up your hands in disgust and decide to shell out to an implementation in a less sugary-wonderful *unsafe* language.

Worse, when you want to talk directly to the operating system, you *have* to talk to an unsafe language: *C*. C is ever-present and unavoidable. It's the lingua-franca of the programming world. Even other safe languages generally expose C interfaces for the world at large! Regardless of why you're doing it, as soon as your program starts talking to C it stops being safe.

With that said, Rust is *totally* a safe programming language.

Well, Rust *has* a safe programming language. Let's step back a bit.

Rust can be thought of as being composed of two programming languages: *Safe Rust* and *Unsafe Rust*. Safe Rust is For Reals Totally Safe. Unsafe Rust, unsurprisingly, is *not* For Reals Totally Safe. In fact, Unsafe Rust lets you do some really crazy unsafe things.

Safe Rust is the *true* Rust programming language. If all you do is write Safe Rust, you will never have to worry about type-safety or memory-safety. You will never endure a null or dangling pointer, or any of that Undefined Behaviour nonsense.

That's totally awesome.

The standard library also gives you enough utilities out-of-the-box that you'll be able to write awesome high-performance applications and libraries in pure idiomatic Safe Rust.

But maybe you want to talk to another language. Maybe you're writing a low-level abstraction not exposed by the standard library. Maybe you're *writing* the standard library (which is written entirely in Rust). Maybe you need to do something the type-system doesn't understand and just *frob some dang bits*. Maybe you need Unsafe Rust.

Unsafe Rust is exactly like Safe Rust with all the same rules and semantics. However Unsafe Rust lets you do some *extra* things that are Definitely Not Safe.

The only things that are different in Unsafe Rust are that you can:

- Dereference raw pointers
- Call `unsafe` functions (including C functions, intrinsics, and the raw allocator)
- Implement `unsafe` traits
- Mutate statics

That's it. The reason these operations are relegated to Unsafe is that misusing any of these things will cause the ever dreaded Undefined Behaviour. Invoking Undefined Behaviour gives the compiler full rights to do arbitrarily bad things to your program. You definitely *should not* invoke Undefined Behaviour.

Unlike C, Undefined Behaviour is pretty limited in scope in Rust. All the core language cares about is preventing the following things:

- Dereferencing null or dangling pointers
- Reading uninitialized memory (chapter 6, page 57)
- Breaking the [pointer aliasing rules]
- Producing invalid primitive values:
  - dangling/null references
  - a `bool` that isn't 0 or 1
  - an undefined `enum` discriminant
  - a `char` outside the ranges [0x0, 0xD7FF] and [0xE000, 0x10FFFF]
  - A non-utf8 `str`
- Unwinding into another language
- Causing a data race (section 9, page 79)

That's it. That's all the causes of Undefined Behaviour baked into Rust. Of course, unsafe functions and traits are free to declare arbitrary other constraints that a program must maintain to avoid Undefined Behaviour. However, generally violations of these constraints will just transitively lead to one of the above problems. Some additional constraints may also derive from compiler intrinsics that make special assumptions about how code can be optimized.

Rust is otherwise quite permissive with respect to other dubious operations. Rust considers it "safe" to:

- Deadlock
- Have a race condition (section 9, page 79)
- Leak memory
- Fail to call destructors
- Overflow integers
- Abort the program
- Delete the production database

However any program that actually manages to do such a thing is *probably* incorrect. Rust provides lots of tools to make these things rare, but these problems are considered impractical to categorically prevent.

How Safe and Unsafe Interact

So what's the relationship between Safe and Unsafe Rust? How do they interact?

Rust models the separation between Safe and Unsafe Rust with the `unsafe` keyword, which can be thought as a sort of *foreign function interface* (FFI) between Safe and Unsafe Rust. This is the magic behind why we can say Safe Rust is a safe language: all the scary unsafe bits are relegated exclusively to FFI *just like every other safe language*.

However because one language is a subset of the other, the two can be cleanly intermixed as long as the boundary between Safe and Unsafe Rust is denoted with the `unsafe` keyword. No need to write headers, initialize runtimes, or any of that other FFI boiler-plate.

There are several places `unsafe` can appear in Rust today, which can largely be grouped into two categories:

- There are unchecked contracts here. To declare you understand this, I require you to write `unsafe` elsewhere:
  - On functions, `unsafe` is declaring the function to be unsafe to call. Users of the function must check the documentation to determine what this means, and then have to write `unsafe` somewhere to identify that they're aware of the danger.
  - On trait declarations, `unsafe` is declaring that *implementing* the trait is an unsafe operation, as it has contracts that other unsafe code is free to trust blindly. (More on this below.)
- I am declaring that I have, to the best of my knowledge, adhered to the unchecked contracts:
  - On trait implementations, `unsafe` is declaring that the contract of the `unsafe` trait has been upheld.
  - On blocks, `unsafe` is declaring any unsafety from an unsafe operation within to be handled, and therefore the parent function is safe.

There is also `#[unsafe_no_drop_flag]`, which is a special case that exists for historical reasons and is in the process of being phased out. See the section on drop flags (section 6, page 59) for details.

Some examples of unsafe functions:

- `slice::get_unchecked` will perform unchecked indexing, allowing memory safety to be freely violated.
- `ptr::offset` is an intrinsic that invokes Undefined Behaviour if it is not "in bounds" as defined by LLVM.
- `mem::transmute` reinterprets some value as having the given type, bypassing type safety in arbitrary ways. (see [conversions] for details)
- All FFI functions are `unsafe` because they can do arbitrary things. C being an obvious culprit, but generally any language can do something that Rust isn't happy about.

As of Rust 1.0 there are exactly two unsafe traits:

- `Send` is a marker trait (it has no actual API) that promises implementors are safe to send (move) to another thread.

- `Sync` is a marker trait that promises that threads can safely share implementors through a shared reference.

The need for unsafe traits boils down to the fundamental property of safe code:

No matter how completely awful Safe code is, it can't cause Undefined Behaviour.

This means that Unsafe Rust, **the royal vanguard of Undefined Behaviour**, has to be *super paranoid* about generic safe code. To be clear, Unsafe Rust is totally free to trust specific safe code. Anything else would degenerate into infinite spirals of paranoid despair. In particular it's generally regarded as ok to trust the standard library to be correct. `std` is effectively an extension of the language, and you really just have to trust the language. If `std` fails to uphold the guarantees it declares, then it's basically a language bug.

That said, it would be best to minimize *needlessly* relying on properties of concrete safe code. Bugs happen! Of course, I must reinforce that this is only a concern for Unsafe code. Safe code can blindly trust anyone and everyone as far as basic memory-safety is concerned.

On the other hand, safe traits are free to declare arbitrary contracts, but because implementing them is safe, unsafe code can't trust those contracts to actually be upheld. This is different from the concrete case because *anyone* can randomly implement the interface. There is something fundamentally different about trusting a particular piece of code to be correct, and trusting *all the code that will ever be written* to be correct.

For instance Rust has `PartialOrd` and `Ord` traits to try to differentiate between types which can "just" be compared, and those that actually implement a total ordering. Pretty much every API that wants to work with data that can be compared wants Ord data. For instance, a sorted map like BTreeMap *doesn't even make sense* for partially ordered types. If you claim to implement Ord for a type, but don't actually provide a proper total ordering, BTreeMap will get *really confused* and start making a total mess of itself. Data that is inserted may be impossible to find!

But that's okay. BTreeMap is safe, so it guarantees that even if you give it a completely garbage Ord implementation, it will still do something *safe*. You won't start reading uninitialized or unallocated memory. In fact, BTreeMap manages to not actually lose any of your data. When the map is dropped, all the destructors will be successfully called! Hooray!

However BTreeMap is implemented using a modest spoonful of Unsafe Rust (most collections are). That means that it's not necessarily *trivially true* that a bad Ord implementation will make BTreeMap behave safely. BTreeMap must be sure not to rely on Ord *where safety is at stake*. Ord is provided by safe code, and safety is not safe code's responsibility to uphold.

But wouldn't it be grand if there was some way for Unsafe to trust some trait contracts *somewhere*? This is the problem that unsafe traits tackle: by marking *the trait itself* as unsafe to implement, unsafe code can trust the implementation to uphold the trait's contract. Although the trait implementation may be incorrect in arbitrary other ways.

For instance, given a hypothetical UnsafeOrd trait, this is technically a valid implementation:

```
unsafe impl UnsafeOrd for MyType {
    fn cmp(&self, other: &Self) -> Ordering {
        Ordering::Equal
    }
}
```

But it's probably not the implementation you want.

Rust has traditionally avoided making traits unsafe because it makes Unsafe pervasive, which is not desirable. The reason Send and Sync are unsafe is because thread safety is a *fundamental property* that unsafe code cannot possibly hope to defend against in the same way it would defend against a bad Ord implementation. The only way to possibly defend against thread-unsafety would be to *not use threading at all*. Making every load and store atomic isn't even sufficient, because it's possible for complex invariants to exist between disjoint locations in memory. For instance, the pointer and capacity of a Vec must be in sync.

Even concurrent paradigms that are traditionally regarded as Totally Safe like message passing implicitly rely on some notion of thread safety – are you really message-passing if you pass a pointer? Send and Sync therefore require some fundamental level of trust that Safe code can't provide, so they must be unsafe to implement. To help obviate the pervasive unsafety that this would introduce, Send (resp. Sync) is automatically derived for all types composed only of Send (resp. Sync) values. 99% of types are Send and Sync, and 99% of those never actually say it (the remaining 1% is overwhelmingly synchronization primitives).

Working with Unsafe

Rust generally only gives us the tools to talk about Unsafe Rust in a scoped and binary manner. Unfortunately, reality is significantly more complicated than that. For instance, consider the following toy function:

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx < arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

Clearly, this function is safe. We check that the index is in bounds, and if it is, index into the array in an unchecked manner. But even in such a trivial function, the scope of the unsafe block is questionable. Consider changing the `<` to a `<=`:

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx <= arr.len() {
        unsafe {
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

This program is now unsound, and yet *we only modified safe code*. This is the fundamental problem of safety: it's non-local. The soundness of our unsafe operations necessarily depends on the state established by otherwise "safe" operations.

Safety is modular in the sense that opting into unsafety doesn't require you to consider arbitrary other kinds of badness. For instance, doing an unchecked index into a slice doesn't mean you suddenly need to worry about the slice being null or containing uninitialized memory. Nothing fundamentally changes. However safety *isn't* modular in the sense that programs are inherently stateful and your unsafe operations may depend on arbitrary other state.

Trickier than that is when we get into actual statefulness. Consider a simple implementation of `Vec`:

```
use std::ptr;

// Note this definition is insufficient. See the section on implementing Vec.
pub struct Vec<T> {
    ptr: *mut T,
    len: usize,
    cap: usize,
}

// Note this implementation does not correctly handle zero-sized types.
// We currently live in a nice imaginary world of only positive fixed-size
// types.
impl<T> Vec<T> {
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap {
            // not important for this example
            self.reallocate();
        }
        unsafe {
            ptr::write(self.ptr.offset(self.len as isize), elem);
            self.len += 1;
        }
    }

    # fn reallocate(&mut self) { }
}
```

This code is simple enough to reasonably audit and verify. Now consider adding the following method:

```
fn make_room(&mut self) {
    // grow the capacity
    self.cap += 1;
}
```

This code is 100% Safe Rust but it is also completely unsound. Changing the capacity violates the invariants of Vec (that `cap` reflects the allocated space in the Vec). This is not something the

rest of Vec can guard against. It *has* to trust the capacity field because there's no way to verify it.

`unsafe` does more than pollute a whole function: it pollutes a whole *module*. Generally, the only bullet-proof way to limit the scope of unsafe code is at the module boundary with privacy.

However this works *perfectly*. The existence of `make_room` is *not* a problem for the soundness of Vec because we didn't mark it as public. Only the module that defines this function can call it. Also, `make_room` directly accesses the private fields of Vec, so it can only be written in the same module as Vec.

It is therefore possible for us to write a completely safe abstraction that relies on complex invariants. This is *critical* to the relationship between Safe Rust and Unsafe Rust. We have already seen that Unsafe code must trust *some* Safe code, but can't trust *generic* Safe code. It can't trust an arbitrary implementor of a trait or any function that was passed to it to be well-behaved in a way that safe code doesn't care about.

However if unsafe code couldn't prevent client safe code from messing with its state in arbitrary ways, safety would be a lost cause. Thankfully, it *can* prevent arbitrary code from messing with critical state due to privacy.

Safety lives!

3

Data Layout

Low-level programming cares a lot about data layout. It's a big deal. It also pervasively influences the rest of the language, so we're going to start by digging into how data is represented in Rust.

repr(Rust)

First and foremost, all types have an alignment specified in bytes. The alignment of a type specifies what addresses are valid to store the value at. A value of alignment `n` must only be stored at an address that is a multiple of `n`. So alignment 2 means you must be stored at an even address, and 1 means that you can be stored anywhere. Alignment is at least 1, and always a power of 2. Most primitives are generally aligned to their size, although this is platform-specific behaviour. In particular, on x86 `u64` and `f64` may be only aligned to 32 bits.

A type's size must always be a multiple of its alignment. This ensures that an array of that type may always be indexed by offsetting by a multiple of its size. Note that the size and alignment of a type may not be known statically in the case of dynamically sized types[1].

Rust gives you the following ways to lay out composite data:

- structs (named product types)
- tuples (anonymous product types)
- arrays (homogeneous product types)
- enums (named sum types – tagged unions)

An enum is said to be *C-like* if none of its variants have associated data.

Composite structures will have an alignment equal to the maximum of their fields' alignment. Rust will consequently insert padding where necessary to ensure that all fields are properly aligned and that the overall type's size is a multiple of its alignment. For instance:

[1] exotic-sizes.html#dynamically-sized-types-(dsts)

```
struct A {
    a: u8,
    b: u32,
    c: u16,
}
```

will be 32-bit aligned on an architecture that aligns these primitives to their respective sizes. The whole struct will therefore have a size that is a multiple of 32-bits. It will potentially become:

```
struct A {
    a: u8,
    _pad1: [u8; 3], // to align `b`
    b: u32,
    c: u16,
    _pad2: [u8; 2], // to make overall size multiple of 4
}
```

There is *no indirection* for these types; all data is stored within the struct, as you would expect in C. However with the exception of arrays (which are densely packed and in-order), the layout of data is not by default specified in Rust. Given the two following struct definitions:

```
struct A {
    a: i32,
    b: u64,
}

struct B {
    a: i32,
    b: u64,
}
```

Rust *does* guarantee that two instances of A have their data laid out in exactly the same way. However Rust *does not* currently guarantee that an instance of A has the same field ordering or padding as an instance of B, though in practice there's no reason why they wouldn't.

With A and B as written, this point would seem to be pedantic, but several other features of Rust make it desirable for the language to play with data layout in complex ways.

For instance, consider this struct:

```
struct Foo<T, U> {
    count: u16,
    data1: T,
    data2: U,
}
```

Now consider the monomorphizations of `Foo<u32, u16>` and `Foo<u16, u32>`. If Rust lays out the fields in the order specified, we expect it to pad the values in the struct to satisfy their alignment requirements. So if Rust didn't reorder fields, we would expect it to produce the following:

```
struct Foo<u16, u32> {
    count: u16,
    data1: u16,
    data2: u32,
}

struct Foo<u32, u16> {
    count: u16,
    _pad1: u16,
    data1: u32,
    data2: u16,
    _pad2: u16,
}
```

The latter case quite simply wastes space. An optimal use of space therefore requires different monomorphizations to have *different field orderings*.

Note: this is a hypothetical optimization that is not yet implemented in Rust 1.0

Enums make this consideration even more complicated. Naively, an enum such as:

```
enum Foo {
    A(u32),
    B(u64),
    C(u8),
}
```

would be laid out as:

```
struct FooRepr {
    data: u64, // this is either a u64, u32, or u8 based on `tag`
    tag: u8,   // 0 = A, 1 = B, 2 = C
}
```

And indeed this is approximately how it would be laid out in general (modulo the size and position of `tag`).

However there are several cases where such a representation is inefficient. The classic case of this is Rust's "null pointer optimization": an enum consisting of a single outer unit variant (e.g. `None`) and a (potentially nested) non- nullable pointer variant (e.g. `&T`) makes the tag unnecessary, because a null pointer value can safely be interpreted to mean that the unit variant is chosen instead. The net result is that, for example, `size_of::<Option<&T>>() == size_of::<&T>()`.

There are many types in Rust that are, or contain, non-nullable pointers such as `Box<T>`, `Vec<T>`, `String`, `&T`, and `&mut T`. Similarly, one can imagine nested enums pooling their tags into a single discriminant, as they are by definition known to have a limited range of valid values. In principle enums could use fairly elaborate algorithms to cache bits throughout nested types with special constrained representations. As such it is *especially* desirable that we leave enum layout unspecified today.

Exotically Sized Types

Most of the time, we think in terms of types with a fixed, positive size. This is not always the case, however.

Dynamically Sized Types (DSTs)

Rust in fact supports Dynamically Sized Types (DSTs): types without a statically known size or alignment. On the surface, this is a bit nonsensical: Rust *must* know the size and alignment of something in order to correctly work with it! In this regard, DSTs are not normal types. Due to their lack of a statically known size, these types can only exist behind some kind of pointer. Any pointer to a DST consequently becomes a *fat* pointer consisting of the pointer and the information that "completes" them (more on this below).

There are two major DSTs exposed by the language: trait objects, and slices.

A trait object represents some type that implements the traits it specifies. The exact original type is *erased* in favour of runtime reflection with a vtable containing all the information necessary to use the type. This is the information that completes a trait object: a pointer to its vtable.

A slice is simply a view into some contiguous storage – typically an array or `Vec`. The information that completes a slice is just the number of elements it points to.

Structs can actually store a single DST directly as their last field, but this makes them a DST as well:

```
// Can't be stored on the stack directly
struct Foo {
    info: u32,
    data: [u8],
}
```

NOTE: As of Rust 1.0 struct DSTs are broken if the last field has a variable position based on its alignment[2].

Zero Sized Types (ZSTs)

Rust actually allows types to be specified that occupy no space:

```
struct Foo; // No fields = no size

// All fields have no size = no size
struct Baz {
    foo: Foo,
    qux: (),      // empty tuple has no size
    baz: [u8; 0], // empty array has no size
}
```

[2] https://github.com/rust-lang/rust/issues/26403

On their own, Zero Sized Types (ZSTs) are, for obvious reasons, pretty useless. However as with many curious layout choices in Rust, their potential is realized in a generic context: Rust largely understands that any operation that produces or stores a ZST can be reduced to a no-op. First off, storing it doesn't even make sense – it doesn't occupy any space. Also there's only one value of that type, so anything that loads it can just produce it from the aether – which is also a no-op since it doesn't occupy any space.

One of the most extreme example's of this is Sets and Maps. Given a `Map<Key, Value>`, it is common to implement a `Set<Key>` as just a thin wrapper around `Map<Key, UselessJunk>`. In many languages, this would necessitate allocating space for UselessJunk and doing work to store and load UselessJunk only to discard it. Proving this unnecessary would be a difficult analysis for the compiler.

However in Rust, we can just say that `Set<Key> = Map<Key, ()>`. Now Rust statically knows that every load and store is useless, and no allocation has any size. The result is that the monomorphized code is basically a custom implementation of a HashSet with none of the overhead that HashMap would have to support values.

Safe code need not worry about ZSTs, but *unsafe* code must be careful about the consequence of types with no size. In particular, pointer offsets are no-ops, and standard allocators (including jemalloc, the one used by default in Rust) may return `nullptr` when a zero-sized allocation is requested, which is indistinguishable from out of memory.

Empty Types

Rust also enables types to be declared that *cannot even be instantiated*. These types can only be talked about at the type level, and never at the value level. Empty types can be declared by specifying an enum with no variants:

```
enum Void {} // No variants = EMPTY
```

Empty types are even more marginal than ZSTs. The primary motivating example for Void types is type-level unreachability. For instance, suppose an API needs to return a Result in general, but a specific case actually is infallible. It's actually possible to communicate this at the type level by returning a `Result<T, Void>`. Consumers of the API can confidently unwrap such a Result knowing that it's *statically impossible* for this value to be an `Err`, as this would require providing a value of type `Void`.

In principle, Rust can do some interesting analyses and optimizations based on this fact. For instance, `Result<T, Void>` could be represented as just `T`, because the `Err` case doesn't actually exist. The following *could* also compile:

```
enum Void {}

let res: Result<u32, Void> = Ok(0);

// Err doesn't exist anymore, so Ok is actually irrefutable.
let Ok(num) = res;
```

But neither of these tricks work today, so all Void types get you is the ability to be confident that certain situations are statically impossible.

One final subtle detail about empty types is that raw pointers to them are actually valid to construct, but dereferencing them is Undefined Behaviour because that doesn't actually make sense. That is, you could model C's `void *` type with `*const Void`, but this doesn't necessarily gain anything over using e.g. `*const ()`, which *is* safe to randomly dereference.

Other reprs

Rust allows you to specify alternative data layout strategies from the default.

repr(C)

This is the most important `repr`. It has fairly simple intent: do what C does. The order, size, and alignment of fields is exactly what you would expect from C or C++. Any type you expect to pass through an FFI boundary should have `repr(C)`, as C is the lingua-franca of the programming world. This is also necessary to soundly do more elaborate tricks with data layout such as reinterpreting values as a different type.

However, the interaction with Rust's more exotic data layout features must be kept in mind. Due to its dual purpose as "for FFI" and "for layout control", `repr(C)` can be applied to types that will be nonsensical or problematic if passed through the FFI boundary.

- ZSTs are still zero-sized, even though this is not a standard behaviour in C, and is explicitly contrary to the behaviour of an empty type in C++, which still consumes a byte of space.
- DSTs, tuples, and tagged unions are not a concept in C and as such are never FFI safe.
- **If the type would have any drop flags (section 6, page 59), they will still be added**
- This is equivalent to one of `repr(u*)` (see the next section) for enums. The chosen size is the default enum size for the target platform's C ABI. Note that enum representation in C is implementation defined, so this is really a "best guess". In particular, this may be incorrect when the C code of interest is compiled with certain flags.

repr(u8), repr(u16), repr(u32), repr(u64)

These specify the size to make a C-like enum. If the discriminant overflows the integer it has to fit in, it will produce a compile-time error. You can manually ask Rust to allow this by setting the overflowing element to explicitly be 0. However Rust will not allow you to create an enum where two variants have the same discriminant.

On non-C-like enums, this will inhibit certain optimizations like the null- pointer optimization.

These reprs have no effect on a struct.

repr(packed)

`repr(packed)` forces rust to strip any padding, and only align the type to a byte. This may improve the memory footprint, but will likely have other negative side-effects.

In particular, most architectures *strongly* prefer values to be aligned. This may mean the unaligned loads are penalized (x86), or even fault (some ARM chips). For simple cases like directly loading or storing a packed field, the compiler might be able to paper over alignment issues with shifts and masks. However if you take a reference to a packed field, it's unlikely that the compiler will be able to emit code to avoid an unaligned load.

As of Rust 1.0 this can cause undefined behaviour.[3]

`repr(packed)` is not to be used lightly. Unless you have extreme requirements, this should not be used.

This repr is a modifier on `repr(C)` and `repr(rust)`.

[3] https://github.com/rust-lang/rust/issues/27060

4

Ownership

Ownership is the breakout feature of Rust. It allows Rust to be completely memory-safe and efficient, while avoiding garbage collection. Before getting into the ownership system in detail, we will consider the motivation of this design.

We will assume that you accept that garbage collection (GC) is not always an optimal solution, and that it is desirable to manually manage memory in some contexts. If you do not accept this, might I interest you in a different language?

Regardless of your feelings on GC, it is pretty clearly a *massive* boon to making code safe. You never have to worry about things going away *too soon* (although whether you still wanted to be pointing at that thing is a different issue…). This is a pervasive problem that C and C++ programs need to deal with. Consider this simple mistake that all of us who have used a non-GC'd language have made at one point:

```
fn as_str(data: &u32) -> &str {
    // compute the string
    let s = format!("{}", data);

    // OH NO! We returned a reference to something that
    // exists only in this function!
    // Dangling pointer! Use after free! Alas!
    // (this does not compile in Rust)
    &s
}
```

This is exactly what Rust's ownership system was built to solve. Rust knows the scope in which the `&s` lives, and as such can prevent it from escaping. However this is a simple case that even a C compiler could plausibly catch. Things get more complicated as code gets bigger and pointers get fed through various functions. Eventually, a C compiler will fall down and won't be able to perform sufficient escape analysis to prove your code unsound. It will consequently be forced to accept your program on the assumption that it is correct.

This will never happen to Rust. It's up to the programmer to prove to the compiler that everything is sound.

Of course, Rust's story around ownership is much more complicated than just verifying that references don't escape the scope of their referent. That's because ensuring pointers are always valid is much more complicated than this. For instance in this code,

```
let mut data = vec![1, 2, 3];
// get an internal reference
let x = &data[0];

// OH NO! `push` causes the backing storage of `data` to be reallocated.
// Dangling pointer! User after free! Alas!
// (this does not compile in Rust)
data.push(4);

println!("{}", x);
```

naive scope analysis would be insufficient to prevent this bug, because `data` does in fact live as long as we needed. However it was *changed* while we had a reference into it. This is why Rust requires any references to freeze the referent and its owners.

References

This section gives a high-level view of the memory model that *all* Rust programs must satisfy to be correct. Safe code is statically verified to obey this model by the borrow checker. Unsafe code may go above and beyond the borrow checker while still satisfying this model. The borrow checker may also be extended to allow more programs to compile, as long as this more fundamental model is satisfied.

There are two kinds of reference:

- Shared reference: `&`
- Mutable reference: `&mut`

Which obey the following rules:

- A reference cannot outlive its referent
- A mutable reference cannot be aliased

That's it. That's the whole model. Of course, we should probably define what *aliased* means. To define aliasing, we must define the notion of *paths* and *liveness*.

NOTE: The model that follows is generally agreed to be dubious and have issues. It's ok-ish as an intuitive model, but fails to capture the desired semantics. We leave this here to be able to use notions introduced here in later sections. This will be significantly changed in the future. TODO: do that.

Paths

If all Rust had were values (no pointers), then every value would be uniquely owned by a variable or composite structure. From this we naturally derive a *tree* of ownership. The stack itself is the root of the tree, with every variable as its direct children. Each variable's direct children would be their fields (if any), and so on.

From this view, every value in Rust has a unique *path* in the tree of ownership. Of particular interest are *ancestors* and *descendants*: if `x` owns `y`, then `x` is an ancestor of `y`, and `y` is a descendant of `x`. Note that this is an inclusive relationship: `x` is a descendant and ancestor of itself.

We can then define references as simply *names* for paths. When you create a reference, you're declaring that an ownership path exists to this address of memory.

Tragically, plenty of data doesn't reside on the stack, and we must also accommodate this. Globals and thread-locals are simple enough to model as residing at the bottom of the stack (though we must be careful with mutable globals). Data on the heap poses a different problem.

If all Rust had on the heap was data uniquely owned by a pointer on the stack, then we could just treat such a pointer as a struct that owns the value on the heap. Box, Vec, String, and HashMap, are examples of types which uniquely own data on the heap.

Unfortunately, data on the heap is not *always* uniquely owned. Rc for instance introduces a notion of *shared* ownership. Shared ownership of a value means there is no unique path to it. A value with no unique path limits what we can do with it.

In general, only shared references can be created to non-unique paths. However mechanisms which ensure mutual exclusion may establish One True Owner temporarily, establishing a unique path to that value (and therefore all its children). If this is done, the value may be mutated. In particular, a mutable reference can be taken.

The most common way to establish such a path is through *interior mutability*, in contrast to the *inherited mutability* that everything in Rust normally uses. Cell, RefCell, Mutex, and RWLock are all examples of interior mutability types. These types provide exclusive access through runtime restrictions.

An interesting case of this effect is Rc itself: if an Rc has refcount 1, then it is safe to mutate or even move its internals. Note however that the refcount itself uses interior mutability.

In order to correctly communicate to the type system that a variable or field of a struct can have interior mutability, it must be wrapped in an UnsafeCell. This does not in itself make it safe to perform interior mutability operations on that value. You still must yourself ensure that mutual exclusion is upheld.

Liveness

Note: Liveness is not the same thing as a *lifetime*, which will be explained in detail in the next section of this chapter.

Roughly, a reference is *live* at some point in a program if it can be dereferenced. Shared references are always live unless they are literally unreachable (for instance, they reside in freed or leaked memory). Mutable references can be reachable but *not* live through the process of *reborrowing*.

A mutable reference can be reborrowed to either a shared or mutable reference to one of its descendants. A reborrowed reference will only be live again once all reborrows derived from it expire. For instance, a mutable reference can be reborrowed to point to a field of its referent:

```
let x = &mut (1, 2);
{
    // reborrow x to a subfield
    let y = &mut x.0;
    // y is now live, but x isn't
    *y = 3;
}
// y goes out of scope, so x is live again
*x = (5, 7);
```

It is also possible to reborrow into *multiple* mutable references, as long as they are *disjoint*: no reference is an ancestor of another. Rust explicitly enables this to be done with disjoint struct fields, because disjointness can be statically proven:

```
let x = &mut (1, 2);
{
    // reborrow x to two disjoint subfields
    let y = &mut x.0;
    let z = &mut x.1;

    // y and z are now live, but x isn't
    *y = 3;
    *z = 4;
}
// y and z go out of scope, so x is live again
*x = (5, 7);
```

However it's often the case that Rust isn't sufficiently smart to prove that multiple borrows are disjoint. *This does not mean it is fundamentally illegal to make such a borrow*, just that Rust isn't as smart as you want.

To simplify things, we can model variables as a fake type of reference: *owned* references. Owned references have much the same semantics as mutable references: they can be re-borrowed in a mutable or shared manner, which makes them no longer live. Live owned references have the unique property that they can be moved out of (though mutable references *can* be swapped out of). This power is only given to *live* owned references because moving its referent would of course invalidate all outstanding references prematurely.

As a local lint against inappropriate mutation, only variables that are marked as `mut` can be borrowed mutably.

It is interesting to note that Box behaves exactly like an owned reference. It can be moved out of, and Rust understands it sufficiently to reason about its paths like a normal variable.

Aliasing

With liveness and paths defined, we can now properly define *aliasing*:

A mutable reference is aliased if there exists another live reference to one of its ancestors or descendants.

(If you prefer, you may also say the two live references alias *each other*. This has no semantic consequences, but is probably a more useful notion when verifying the soundness of a construct.)

That's it. Super simple right? Except for the fact that it took us two pages to define all of the terms in that definition. You know: Super. Simple.

Actually it's a bit more complicated than that. In addition to references, Rust has *raw pointers*: `*const T` and `*mut T`. Raw pointers have no inherent ownership or aliasing semantics. As a result, Rust makes absolutely no effort to track that they are used correctly, and they are wildly unsafe.

It is an open question to what degree raw pointers have alias semantics. However it is important for these definitions to be sound that the existence of a raw pointer does not imply some kind of live path.

Lifetimes

Rust enforces these rules through *lifetimes*. Lifetimes are effectively just names for scopes somewhere in the program. Each reference, and anything that contains a reference, is tagged with a lifetime specifying the scope it's valid for.

Within a function body, Rust generally doesn't let you explicitly name the lifetimes involved. This is because it's generally not really necessary to talk about lifetimes in a local context; Rust has all the information and can work out everything as optimally as possible. Many anonymous scopes and temporaries that you would otherwise have to write are often introduced to make your code Just Work.

However once you cross the function boundary, you need to start talking about lifetimes. Lifetimes are denoted with an apostrophe: `'a`, `'static`. To dip our toes with lifetimes, we're going to pretend that we're actually allowed to label scopes with lifetimes, and desugar the examples from the start of this chapter.

Originally, our examples made use of *aggressive* sugar – high fructose corn syrup even – around scopes and lifetimes, because writing everything out explicitly is *extremely noisy*. All Rust code relies on aggressive inference and elision of "obvious" things.

One particularly interesting piece of sugar is that each `let` statement implicitly introduces a scope. For the most part, this doesn't really matter. However it does matter for variables that refer to each other. As a simple example, let's completely desugar this simple piece of Rust code:

```
let x = 0;
let y = &x;
let z = &y;
```

The borrow checker always tries to minimize the extent of a lifetime, so it will likely desugar to the following:

```
// NOTE: `'a: {` and `&'b x` is not valid syntax!
'a: {
    let x: i32 = 0;
    'b: {
        // lifetime used is 'b because that's good enough.
        let y: &'b i32 = &'b x;
        'c: {
            // ditto on 'c
            let z: &'c &'b i32 = &'c y;
        }
    }
}
```

Wow. That's… awful. Let's all take a moment to thank Rust for making this easier.

Actually passing references to outer scopes will cause Rust to infer a larger lifetime:

```
let x = 0;
let z;
let y = &x;
z = y;
```

```
'a: {
    let x: i32 = 0;
    'b: {
        let z: &'b i32;
        'c: {
            // Must use 'b here because this reference is
            // being passed to that scope.
            let y: &'b i32 = &'b x;
            z = y;
        }
    }
}
```

Example: references that outlive referents

Alright, let's look at some of those examples from before:

```
fn as_str(data: &u32) -> &str {
    let s = format!("{}", data);
    &s
}
```

desugars to:

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s;
    }
}
```

This signature of `as_str` takes a reference to a u32 with *some* lifetime, and promises that it can produce a reference to a str that can live *just as long*. Already we can see why this signature might be trouble. That basically implies that we're going to find a str somewhere in the scope the reference to the u32 originated in, or somewhere *even earlier*. That's a bit of a big ask.

We then proceed to compute the string `s`, and return a reference to it. Since the contract of our function says the reference must outlive `'a`, that's the lifetime we infer for the reference. Unfortunately, `s` was defined in the scope `'b`, so the only way this is sound is if `'b` contains `'a` – which is clearly false since `'a` must contain the function call itself. We have therefore created a reference whose lifetime outlives its referent, which is *literally* the first thing we said that references can't do. The compiler rightfully blows up in our face.

To make this more clear, we can expand the example:

```
fn as_str<'a>(data: &'a u32) -> &'a str {
    'b: {
        let s = format!("{}", data);
        return &'a s
    }
}

fn main() {
    'c: {
        let x: u32 = 0;
        'd: {
            // An anonymous scope is introduced because the borrow does not
            // need to last for the whole scope x is valid for. The return
            // of as_str must find a str somewhere before this function
            // call. Obviously not happening.
            println!("{}", as_str::<'d>(&'d x));
        }
    }
}
```

Shoot!

Of course, the right way to write this function is as follows:

```
fn to_string(data: &u32) -> String {
    format!("{}", data)
}
```

We must produce an owned value inside the function to return it! The only way we could have returned an `&'a str` would have been if it was in a field of the `&'a u32`, which is obviously not the case.

(Actually we could have also just returned a string literal, which as a global can be considered to reside at the bottom of the stack; though this limits our implementation *just a bit*.)

Example: aliasing a mutable reference

How about the other example:

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

```
'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {
        // 'b is as big as we need this borrow to be
        // (just need to get to `println!`)
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            // Temporary scope because we don't need the
            // &mut to last any longer.
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

The problem here is is bit more subtle and interesting. We want Rust to reject this program for the following reason: We have a live shared reference `x` to a descendent of `data` when we try to take a mutable reference to `data` to `push`. This would create an aliased mutable reference, which would violate the *second* rule of references.

However this is *not at all* how Rust reasons that this program is bad. Rust doesn't understand that `x` is a reference to a subpath of `data`. It doesn't understand Vec at all. What it *does* see is that `x` has to live for `'b` to be printed. The signature of `Index::index` subsequently demands that the reference we take to `data` has to survive for `'b`. When we try to call `push`, it then sees us try to make an `&'c mut data`. Rust knows that `'c` is contained within `'b`, and rejects our program because the `&'b data` must still be live!

Here we see that the lifetime system is much more coarse than the reference semantics we're actually interested in preserving. For the most part, *that's totally ok*, because it keeps us from spending all day explaining our program to the compiler. However it does mean that several programs that are totally correct with respect to Rust's *true* semantics are rejected because lifetimes are too dumb.

Limits of Lifetimes

Given the following code:

```
struct Foo;

impl Foo {
    fn mutate_and_share(&mut self) -> &Self { &*self }
    fn share(&self) {}
}

fn main() {
    let mut foo = Foo;
    let loan = foo.mutate_and_share();
    foo.share();
}
```

One might expect it to compile. We call `mutate_and_share`, which mutably borrows `foo` temporarily, but then returns only a shared reference. Therefore we would expect `foo.share()` to succeed as `foo` shouldn't be mutably borrowed.

However when we try to compile it:

```
<anon>:11:5: 11:8 error: cannot borrow `foo` as immutable because it is also borrowed
↳ as mutable
<anon>:11     foo.share();
              ^~~
<anon>:10:16: 10:19 note: previous borrow of `foo` occurs here; the mutable borrow pre
↳ vents subsequent moves, borrows, or modification of `foo` until the borrow ends
<anon>:10     let loan = foo.mutate_and_share();
                         ^~~
<anon>:12:2: 12:2 note: previous borrow ends here
<anon>:8 fn main() {
<anon>:9     let mut foo = Foo;
<anon>:10     let loan = foo.mutate_and_share();
<anon>:11     foo.share();
<anon>:12 }
          ^
```

What happened? Well, we got the exact same reasoning as we did for Example 2 in the previous section[1]. We desugar the program and we get the following:

```
struct Foo;

impl Foo {
    fn mutate_and_share<'a>(&'a mut self) -> &'a Self { &'a *self }
    fn share<'a>(&'a self) {}
```

[1] lifetimes.html#example-2:-aliasing-a-mutable-reference

```
}

fn main() {
    'b: {
        let mut foo: Foo = Foo;
        'c: {
            let loan: &'c Foo = Foo::mutate_and_share::<'c>(&'c mut foo);
            'd: {
                Foo::share::<'d>(&'d foo);
            }
        }
    }
}
```

The lifetime system is forced to extend the `&mut foo` to have lifetime `'c`, due to the lifetime of `loan` and mutate_and_share's signature. Then when we try to call `share`, and it sees we're trying to alias that `&'c mut foo` and blows up in our face!

This program is clearly correct according to the reference semantics we actually care about, but the lifetime system is too coarse-grained to handle that.

TODO: other common problems? SEME regions stuff, mostly?

Lifetime Elision

In order to make common patterns more ergonomic, Rust allows lifetimes to be *elided* in function signatures.

A *lifetime position* is anywhere you can write a lifetime in a type:

```
&'a T
&'a mut T
T<'a>
```

Lifetime positions can appear as either "input" or "output":

- For `fn` definitions, input refers to the types of the formal arguments in the `fn` definition, while output refers to result types. So `fn foo(s: &str) -> (&str, &str)` has elided one lifetime in input position and two lifetimes in output position. Note that the input positions of a `fn` method definition do not include the lifetimes that occur in the method's `impl` header (nor lifetimes that occur in the trait header, for a default method).
- In the future, it should be possible to elide `impl` headers in the same manner.

Elision rules are as follows:

- Each elided lifetime in input position becomes a distinct lifetime parameter.

- If there is exactly one input lifetime position (elided or not), that lifetime is assigned to *all* elided output lifetimes.
- If there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to *all* elided output lifetimes.
- Otherwise, it is an error to elide an output lifetime.

Examples:

```
fn print(s: &str);                                    // elided
fn print<'a>(s: &'a str);                             // expanded

fn debug(lvl: uint, s: &str);                         // elided
fn debug<'a>(lvl: uint, s: &'a str);                  // expanded

fn substr(s: &str, until: uint) -> &str;              // elided
fn substr<'a>(s: &'a str, until: uint) -> &'a str;    // expanded

fn get_str() -> &str;                                 // ILLEGAL

fn frob(s: &str, t: &str) -> &str;                    // ILLEGAL

fn get_mut(&mut self) -> &mut T;                      // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T;            // expanded

fn args<T:ToCStr>(&mut self, args: &[T]) -> &mut Command                  // elided
fn args<'a, 'b, T:ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command // expanded

fn new(buf: &mut [u8]) -> BufWriter;                  // elided
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>        // expanded
```

Unbounded Lifetimes

Unsafe code can often end up producing references or lifetimes out of thin air. Such lifetimes come into the world as *unbounded*. The most common source of this is derefencing a raw pointer, which produces a reference with an unbounded lifetime. Such a lifetime becomes as big as context demands. This is in fact more powerful than simply becoming `'static`, because for instance `&'static &'a T` will fail to typecheck, but the unbound lifetime will perfectly mold into `&'a &'a T` as needed. However for most intents and purposes, such an unbounded lifetime can be regarded as `'static`.

Almost no reference is `'static`, so this is probably wrong. `transmute` and `transmute_copy` are the two other primary offenders. One should endeavour to bound an unbounded lifetime as quick as possible, especially across function boundaries.

Given a function, any output lifetimes that don't derive from inputs are unbounded. For instance:

```
fn get_str<'a>() -> &'a str;
```

will produce an `&str` with an unbounded lifetime. The easiest way to avoid unbounded lifetimes is to use lifetime elision at the function boundary. If an output lifetime is elided, then it *must* be bounded by an input lifetime. Of course it might be bounded by the *wrong* lifetime, but this will usually just cause a compiler error, rather than allow memory safety to be trivially violated.

Within a function, bounding lifetimes is more error-prone. The safest and easiest way to bound a lifetime is to return it from a function with a bound lifetime. However if this is unacceptable, the reference can be placed in a location with a specific lifetime. Unfortunately it's impossible to name all lifetimes involved in a function. To get around this, you can in principle use `copy_lifetime`, though these are unstable due to their awkward nature and questionable utility.

Higher-Rank Trait Bounds

Rust's `Fn` traits are a little bit magic. For instance, we can write the following code:

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    where F: Fn(&(u8, u16)) -> &u8,
{
    fn call(&self) -> &u8 {
        (self.func)(&self.data)
    }
}

fn do_it(data: &(u8, u16)) -> &u8 { &data.0 }

fn main() {
    let clo = Closure { data: (0, 1), func: do_it };
    println!("{}", clo.call());
}
```

If we try to naively desugar this code in the same way that we did in the lifetimes section, we run into some trouble:

```
struct Closure<F> {
    data: (u8, u16),
    func: F,
}

impl<F> Closure<F>
    // where F: Fn(&'??? (u8, u16)) -> &'??? u8,
```

```
{
    fn call<'a>(&'a self) -> &'a u8 {
        (self.func)(&self.data)
    }
}

fn do_it<'b>(data: &'b (u8, u16)) -> &'b u8 { &'b data.0 }

fn main() {
    'x: {
        let clo = Closure { data: (0, 1), func: do_it };
        println!("{}", clo.call());
    }
}
```

How on earth are we supposed to express the lifetimes on `F`'s trait bound? We need to provide some lifetime there, but the lifetime we care about can't be named until we enter the body of `call`! Also, that isn't some fixed lifetime; `call` works with *any* lifetime `&self` happens to have at that point.

This job requires The Magic of Higher-Rank Trait Bounds (HRTBs). The way we desugar this is as follows:

```
where for<'a> F: Fn(&'a (u8, u16)) -> &'a u8,
```

(Where `Fn(a, b, c) -> d` is itself just sugar for the unstable *real* `Fn` trait)

`for<'a>` can be read as "for all choices of `'a`", and basically produces an *infinite list* of trait bounds that F must satisfy. Intense. There aren't many places outside of the `Fn` traits where we encounter HRTBs, and even for those we have a nice magic sugar for the common cases.

Subtyping and Variance

Although Rust doesn't have any notion of structural inheritance, it *does* include subtyping. In Rust, subtyping derives entirely from lifetimes. Since lifetimes are scopes, we can partially order them based on the *contains* (outlives) relationship. We can even express this as a generic bound.

Subtyping on lifetimes is in terms of that relationship: if `'a: 'b` ("a contains b" or "a outlives b"), then `'a` is a subtype of `'b`. This is a large source of confusion, because it seems intuitively backwards to many: the bigger scope is a *subtype* of the smaller scope.

This does in fact make sense, though. The intuitive reason for this is that if you expect an `&'a u8`, then it's totally fine for me to hand you an `&'static u8`, in the same way that if you expect an Animal in Java, it's totally fine for me to hand you a Cat. Cats are just Animals *and more*, just as `'static` is just `'a` *and more*.

(Note, the subtyping relationship and typed-ness of lifetimes is a fairly arbitrary construct that some disagree with. However it simplifies our analysis to treat lifetimes and types uniformly.)

Higher-ranked lifetimes are also subtypes of every concrete lifetime. This is because taking an arbitrary lifetime is strictly more general than taking a specific one.

Variance

Variance is where things get a bit complicated.

Variance is a property that *type constructors* have with respect to their arguments. A type constructor in Rust is a generic type with unbound arguments. For instance `Vec` is a type constructor that takes a `T` and returns a `Vec<T>`. `&` and `&mut` are type constructors that take two inputs: a lifetime, and a type to point to.

A type constructor's *variance* is how the subtyping of its inputs affects the subtyping of its outputs. There are two kinds of variance in Rust:

- F is *variant* over `T` if `T` being a subtype of `U` implies `F<T>` is a subtype of `F<U>` (subtyping "passes through")
- F is *invariant* over `T` otherwise (no subtyping relation can be derived)

(For those of you who are familiar with variance from other languages, what we refer to as "just" variance is in fact *covariance*. Rust does not have contravariance. Historically Rust did have some contravariance but it was scrapped due to poor interactions with other features. If you experience contravariance in Rust call your local compiler developer for medical advice.)

Some important variances:

- `&'a T` is variant over `'a` and `T` (as is `*const T` by metaphor)
- `&'a mut T` is variant with over `'a` but invariant over `T`
- `Fn(T) -> U` is invariant over `T`, but variant over `U`
- `Box`, `Vec`, and all other collections are variant over the types of their contents
- `UnsafeCell<T>`, `Cell<T>`, `RefCell<T>`, `Mutex<T>` and all other interior mutability types are invariant over T (as is `*mut T` by metaphor)

To understand why these variances are correct and desirable, we will consider several examples.

We have already covered why `&'a T` should be variant over `'a` when introducing subtyping: it's desirable to be able to pass longer-lived things where shorter-lived things are needed.

Similar reasoning applies to why it should be variant over T. It is reasonable to be able to pass `&&'static str` where an `&&'a str` is expected. The additional level of indirection does not change the desire to be able to pass longer lived things where shorted lived things are expected.

However this logic doesn't apply to `&mut`. To see why `&mut` should be invariant over T, consider the following code:

```
fn overwrite<T: Copy>(input: &mut T, new: &mut T) {
    *input = *new;
}

fn main() {
    let mut forever_str: &'static str = "hello";
    {
        let string = String::from("world");
        overwrite(&mut forever_str, &mut &*string);
```

```
    }
    // Oops, printing free'd memory
    println!("{}", forever_str);
}
```

The signature of `overwrite` is clearly valid: it takes mutable references to two values of the same type, and overwrites one with the other. If `&mut T` was variant over T, then `&mut &'static str` would be a subtype of `&mut &'a str`, since `&'static str` is a subtype of `&'a str`. Therefore the lifetime of `forever_str` would successfully be "shrunk" down to the shorter lifetime of `string`, and `overwrite` would be called successfully. `string` would subsequently be dropped, and `forever_str` would point to freed memory when we print it! Therefore `&mut` should be invariant.

This is the general theme of variance vs invariance: if variance would allow you to store a short-lived value into a longer-lived slot, then you must be invariant.

However it *is* sound for `&'a mut T` to be variant over `'a`. The key difference between `'a` and T is that `'a` is a property of the reference itself, while T is something the reference is borrowing. If you change T's type, then the source still remembers the original type. However if you change the lifetime's type, no one but the reference knows this information, so it's fine. Put another way: `&'a mut T` owns `'a`, but only *borrows* T.

`Box` and `Vec` are interesting cases because they're variant, but you can definitely store values in them! This is where Rust gets really clever: it's fine for them to be variant because you can only store values in them *via a mutable reference*! The mutable reference makes the whole type invariant, and therefore prevents you from smuggling a short-lived type into them.

Being variant allows `Box` and `Vec` to be weakened when shared immutably. So you can pass a `&Box<&'static str>` where a `&Box<&'a str>` is expected.

However what should happen when passing *by-value* is less obvious. It turns out that, yes, you can use subtyping when passing by-value. That is, this works:

```
fn get_box<'a>(str: &'a str) -> Box<&'a str> {
    // string literals are `&'static str`s
    Box::new("hello")
}
```

Weakening when you pass by-value is fine because there's no one else who "remembers" the old lifetime in the Box. The reason a variant `&mut` was trouble was because there's always someone else who remembers the original subtype: the actual owner.

The invariance of the cell types can be seen as follows: `&` is like an `&mut` for a cell, because you can still store values in them through an `&`. Therefore cells must be invariant to avoid lifetime smuggling.

`Fn` is the most subtle case because it has mixed variance. To see why `Fn(T) -> U` should be invariant over T, consider the following function signature:

```
// 'a is derived from some parent scope
fn foo(&'a str) -> usize;
```

This signature claims that it can handle any `&str` that lives at least as long as `'a`. Now if this signature was variant over `&'a str`, that would mean

```
fn foo(&'static str) -> usize;
```

could be provided in its place, as it would be a subtype. However this function has a stronger requirement: it says that it can only handle `&'static str`s, and nothing else. Giving `&'a str`s to it would be unsound, as it's free to assume that what it's given lives forever. Therefore functions are not variant over their arguments.

To see why `Fn(T) -> U` should be variant over U, consider the following function signature:

```
// 'a is derived from some parent scope
fn foo(usize) -> &'a str;
```

This signature claims that it will return something that outlives `'a`. It is therefore completely reasonable to provide

```
fn foo(usize) -> &'static str;
```

in its place. Therefore functions are variant over their return type.

`*const` has the exact same semantics as `&`, so variance follows. `*mut` on the other hand can dereference to an `&mut` whether shared or not, so it is marked as invariant just like cells.

This is all well and good for the types the standard library provides, but how is variance determined for type that *you* define? A struct, informally speaking, inherits the variance of its fields. If a struct `Foo` has a generic argument `A` that is used in a field `a`, then Foo's variance over `A` is exactly `a`'s variance. However this is complicated if `A` is used in multiple fields.

- If all uses of A are variant, then Foo is variant over A
- Otherwise, Foo is invariant over A

```
use std::cell::Cell;

struct Foo<'a, 'b, A: 'a, B: 'b, C, D, E, F, G, H> {
    a: &'a A,     // variant over 'a and A
    b: &'b mut B, // invariant over 'b and B
    c: *const C,  // variant over C
    d: *mut D,    // invariant over D
    e: Vec<E>,    // variant over E
    f: Cell<F>,   // invariant over F
    g: G,         // variant over G
    h1: H,        // would also be variant over H except...
    h2: Cell<H>,  // invariant over H, because invariance wins
}
```

Drop Check

We have seen how lifetimes provide us some fairly simple rules for ensuring that we never read dangling references. However up to this point we have only ever interacted with the *outlives*

relationship in an inclusive manner. That is, when we talked about `'a: 'b`, it was ok for `'a` to live *exactly* as long as `'b`. At first glance, this seems to be a meaningless distinction. Nothing ever gets dropped at the same time as another, right? This is why we used the following desugarring of `let` statements:

```
let x;
let y;
```

```
{
    let x;
    {
        let y;
    }
}
```

Each creates its own scope, clearly establishing that one drops before the other. However, what if we do the following?

```
let (x, y) = (vec![], vec![]);
```

Does either value strictly outlive the other? The answer is in fact *no*, neither value strictly outlives the other. Of course, one of x or y will be dropped before the other, but the actual order is not specified. Tuples aren't special in this regard; composite structures just don't guarantee their destruction order as of Rust 1.0.

We *could* specify this for the fields of built-in composites like tuples and structs. However, what about something like Vec? Vec has to manually drop its elements via pure-library code. In general, anything that implements Drop has a chance to fiddle with its innards during its final death knell. Therefore the compiler can't sufficiently reason about the actual destruction order of the contents of any type that implements Drop.

So why do we care? We care because if the type system isn't careful, it could accidentally make dangling pointers. Consider the following simple program:

```
struct Inspector<'a>(&'a u8);

fn main() {
    let (inspector, days);
    days = Box::new(1);
    inspector = Inspector(&days);
}
```

This program is totally sound and compiles today. The fact that `days` does not *strictly* outlive `inspector` doesn't matter. As long as the `inspector` is alive, so is days.

However if we add a destructor, the program will no longer compile!

```
struct Inspector<'a>(&'a u8);
```

```
impl<'a> Drop for Inspector<'a> {
    fn drop(&mut self) {
        println!("I was only {} days from retirement!", self.0);
    }
}

fn main() {
    let (inspector, days);
    days = Box::new(1);
    inspector = Inspector(&days);
    // Let's say `days` happens to get dropped first.
    // Then when Inspector is dropped, it will try to read free'd memory!
}
```

```
<anon>:12:28: 12:32 error: `days` does not live long enough
<anon>:12     inspector = Inspector(&days);
                                    ^~~~
<anon>:9:11: 15:2 note: reference must be valid for the block at 9:10...
<anon>:9 fn main() {
<anon>:10     let (inspector, days);
<anon>:11     days = Box::new(1);
<anon>:12     inspector = Inspector(&days);
<anon>:13     // Let's say `days` happens to get dropped first.
<anon>:14     // Then when Inspector is dropped, it will try to read free'd memory!
          ...
<anon>:10:27: 15:2 note: ...but borrowed value is only valid for the block suffix foll
↳ owing statement 0 at 10:26
<anon>:10     let (inspector, days);
<anon>:11     days = Box::new(1);
<anon>:12     inspector = Inspector(&days);
<anon>:13     // Let's say `days` happens to get dropped first.
<anon>:14     // Then when Inspector is dropped, it will try to read free'd memory!
<anon>:15 }
```

Implementing Drop lets the Inspector execute some arbitrary code during its death. This means it can potentially observe that types that are supposed to live as long as it does actually were destroyed first.

Interestingly, only generic types need to worry about this. If they aren't generic, then the only lifetimes they can harbor are `'static`, which will truly live *forever*. This is why this problem is referred to as *sound generic drop*. Sound generic drop is enforced by the *drop checker*. As of this writing, some of the finer details of how the drop checker validates types is totally up in the air. However The Big Rule is the subtlety that we have focused on this whole section:

For a generic type to soundly implement drop, its generics arguments must strictly outlive it.

This rule is sufficient but not necessary to satisfy the drop checker. That is, if your type obeys this rule then it's definitely sound to drop. However there are special cases where you can fail to

satisfy this, but still successfully pass the borrow checker. These are the precise rules that are currently up in the air.

It turns out that when writing unsafe code, we generally don't need to worry at all about doing the right thing for the drop checker. However there is one special case that you need to worry about, which we will look at in the next section.

PhantomData

When working with unsafe code, we can often end up in a situation where types or lifetimes are logically associated with a struct, but not actually part of a field. This most commonly occurs with lifetimes. For instance, the `Iter` for `&'a [T]` is (approximately) defined as follows:

```
struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
}
```

However because `'a` is unused within the struct's body, it's *unbounded*. Because of the troubles this has historically caused, unbounded lifetimes and types are *forbidden* in struct definitions. Therefore we must somehow refer to these types in the body. Correctly doing this is necessary to have correct variance and drop checking.

We do this using `PhantomData`, which is a special marker type. `PhantomData` consumes no space, but simulates a field of the given type for the purpose of static analysis. This was deemed to be less error-prone than explicitly telling the type-system the kind of variance that you want, while also providing other useful such as the information needed by drop check.

Iter logically contains a bunch of `&'a T`s, so this is exactly what we tell the PhantomData to simulate:

```
use std::marker;

struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
    _marker: marker::PhantomData<&'a T>,
}
```

and that's it. The lifetime will be bounded, and your iterator will be variant over `'a` and `T`. Everything Just Works.

Another important example is Vec, which is (approximately) defined as follows:

```
struct Vec<T> {
    data: *const T, // *const for variance!
    len: usize,
    cap: usize,
}
```

Unlike the previous example it *appears* that everything is exactly as we want. Every generic argument to Vec shows up in the at least one field. Good to go!

Nope.

The drop checker will generously determine that Vec does not own any values of type T. This will in turn make it conclude that it doesn't need to worry about Vec dropping any T's in its destructor for determining drop check soundness. This will in turn allow people to create unsoundness using Vec's destructor.

In order to tell dropck that we *do* own values of type T, and therefore may drop some T's when *we* drop, we must add an extra PhantomData saying exactly that:

```
use std::marker;

struct Vec<T> {
    data: *const T, // *const for covariance!
    len: usize,
    cap: usize,
    _marker: marker::PhantomData<T>,
}
```

Raw pointers that own an allocation is such a pervasive pattern that the standard library made a utility for itself called `Unique<T>` which:

- wraps a `*const T` for variance
- includes a `PhantomData<T>`,
- auto-derives Send/Sync as if T was contained
- marks the pointer as NonZero for the null-pointer optimization

Splitting Borrows

The mutual exclusion property of mutable references can be very limiting when working with a composite structure. The borrow checker understands some basic stuff, but will fall over pretty easily. It does understand structs sufficiently to know that it's possible to borrow disjoint fields of a struct simultaneously. So this works today:

```
struct Foo {
    a: i32,
    b: i32,
    c: i32,
}

let mut x = Foo {a: 0, b: 0, c: 0};
let a = &mut x.a;
let b = &mut x.b;
let c = &x.c;
*b += 1;
```

```
let c2 = &x.c;
*a += 10;
println!("{} {} {} {}", a, b, c, c2);
```

However borrowck doesn't understand arrays or slices in any way, so this doesn't work:

```
let mut x = [1, 2, 3];
let a = &mut x[0];
let b = &mut x[1];
println!("{} {}", a, b);
```

```
<anon>:4:14: 4:18 error: cannot borrow `x[..]` as mutable more than once at a time
<anon>:4 let b = &mut x[1];
                      ^~~~
<anon>:3:14: 3:18 note: previous borrow of `x[..]` occurs here; the mutable borrow pre
↳ vents subsequent moves, borrows, or modification of `x[..]` until the borrow ends
<anon>:3 let a = &mut x[0];
                      ^~~~
<anon>:6:2: 6:2 note: previous borrow ends here
<anon>:1 fn main() {
<anon>:2 let mut x = [1, 2, 3];
<anon>:3 let a = &mut x[0];
<anon>:4 let b = &mut x[1];
<anon>:5 println!("{} {}", a, b);
<anon>:6 }
         ^
error: aborting due to 2 previous errors
```

While it was plausible that borrowck could understand this simple case, it's pretty clearly hopeless for borrowck to understand disjointness in general container types like a tree, especially if distinct keys actually *do* map to the same value.

In order to "teach" borrowck that what we're doing is ok, we need to drop down to unsafe code. For instance, mutable slices expose a `split_at_mut` function that consumes the slice and returns two mutable slices. One for everything to the left of the index, and one for everything to the right. Intuitively we know this is safe because the slices don't overlap, and therefore alias. However the implementation requires some unsafety:

```
fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T]) {
    let len = self.len();
    let ptr = self.as_mut_ptr();
    assert!(mid <= len);
    unsafe {
        (from_raw_parts_mut(ptr, mid),
         from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

This is actually a bit subtle. So as to avoid ever making two `&mut`'s to the same value, we explicitly construct brand-new slices through raw pointers.

However more subtle is how iterators that yield mutable references work. The iterator trait is defined as follows:

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

Given this definition, Self::Item has *no* connection to `self`. This means that we can call `next` several times in a row, and hold onto all the results *concurrently*. This is perfectly fine for by-value iterators, which have exactly these semantics. It's also actually fine for shared references, as they admit arbitrarily many references to the same thing (although the iterator needs to be a separate object from the thing being shared).

But mutable references make this a mess. At first glance, they might seem completely incompatible with this API, as it would produce multiple mutable references to the same object!

However it actually *does* work, exactly because iterators are one-shot objects. Everything an IterMut yields will be yielded at most once, so we don't actually ever yield multiple mutable references to the same piece of data.

Perhaps surprisingly, mutable iterators don't require unsafe code to be implemented for many types!

For instance here's a singly linked list:

```
type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    next: Link<T>,
}

pub struct LinkedList<T> {
    head: Link<T>,
}

pub struct IterMut<'a, T: 'a>(Option<&'a mut Node<T>>);

impl<T> LinkedList<T> {
    fn iter_mut(&mut self) -> IterMut<T> {
        IterMut(self.head.as_mut().map(|node| &mut **node))
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
```

```
    fn next(&mut self) -> Option<Self::Item> {
        self.0.take().map(|node| {
            self.0 = node.next.as_mut().map(|node| &mut **node);
            &mut node.elem
        })
    }
}
```

Here's a mutable slice:

```
use std::mem;

pub struct IterMut<'a, T: 'a>(&'a mut[T]);

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;

    fn next(&mut self) -> Option<Self::Item> {
        let slice = mem::replace(&mut self.0, &mut []);
        if slice.is_empty() { return None; }

        let (l, r) = slice.split_at_mut(1);
        self.0 = r;
        l.get_mut(0)
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        let slice = mem::replace(&mut self.0, &mut []);
        if slice.is_empty() { return None; }

        let new_len = slice.len() - 1;
        let (l, r) = slice.split_at_mut(new_len);
        self.0 = l;
        r.get_mut(0)
    }
}
```

And here's a binary tree:

```
use std::collections::VecDeque;

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
    elem: T,
    left: Link<T>,
```

```
    right: Link<T>,
}

pub struct Tree<T> {
    root: Link<T>,
}

struct NodeIterMut<'a, T: 'a> {
    elem: Option<&'a mut T>,
    left: Option<&'a mut Node<T>>,
    right: Option<&'a mut Node<T>>,
}

enum State<'a, T: 'a> {
    Elem(&'a mut T),
    Node(&'a mut Node<T>),
}

pub struct IterMut<'a, T: 'a>(VecDeque<NodeIterMut<'a, T>>);

impl<T> Tree<T> {
    pub fn iter_mut(&mut self) -> IterMut<T> {
        let mut deque = VecDeque::new();
        self.root.as_mut().map(|root| deque.push_front(root.iter_mut()));
        IterMut(deque)
    }
}

impl<T> Node<T> {
    pub fn iter_mut(&mut self) -> NodeIterMut<T> {
        NodeIterMut {
            elem: Some(&mut self.elem),
            left: self.left.as_mut().map(|node| &mut **node),
            right: self.right.as_mut().map(|node| &mut **node),
        }
    }
}


impl<'a, T> Iterator for NodeIterMut<'a, T> {
    type Item = State<'a, T>;

    fn next(&mut self) -> Option<Self::Item> {
        match self.left.take() {
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.right.take() {
```

```
                    Some(node) => Some(State::Node(node)),
                    None => None,
                }
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for NodeIterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        match self.right.take() {
            Some(node) => Some(State::Node(node)),
            None => match self.elem.take() {
                Some(elem) => Some(State::Elem(elem)),
                None => match self.left.take() {
                    Some(node) => Some(State::Node(node)),
                    None => None,
                }
            }
        }
    }
}

impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.front_mut().and_then(|node_it| node_it.next()) {
                Some(State::Elem(elem)) => return Some(elem),
                Some(State::Node(node)) => self.0.push_front(node.iter_mut()),
                None => if let None = self.0.pop_front() { return None },
            }
        }
    }
}

impl<'a, T> DoubleEndedIterator for IterMut<'a, T> {
    fn next_back(&mut self) -> Option<Self::Item> {
        loop {
            match self.0.back_mut().and_then(|node_it| node_it.next_back()) {
                Some(State::Elem(elem)) => return Some(elem),
                Some(State::Node(node)) => self.0.push_back(node.iter_mut()),
                None => if let None = self.0.pop_back() { return None },
            }
        }
    }
}
```

All of these are completely safe and work on stable Rust! This ultimately falls out of the simple

struct case we saw before: Rust understands that you can safely split a mutable reference into subfields. We can then encode permanently consuming a reference via Options (or in the case of slices, replacing with an empty slice).

5

Type Conversions

At the end of the day, everything is just a pile of bits somewhere, and type systems are just there to help us use those bits right. There are two common problems with typing bits: needing to reinterpret those exact bits as a different type, and needing to change the bits to have equivalent meaning for a different type. Because Rust encourages encoding important properties in the type system, these problems are incredibly pervasive. As such, Rust consequently gives you several ways to solve them.

First we'll look at the ways that Safe Rust gives you to reinterpret values. The most trivial way to do this is to just destructure a value into its constituent parts and then build a new type out of them. e.g.

```
struct Foo {
    x: u32,
    y: u16,
}

struct Bar {
    a: u32,
    b: u16,
}

fn reinterpret(foo: Foo) -> Bar {
    let Foo { x, y } = foo;
    Bar { a: x, b: y }
}
```

But this is, at best, annoying. For common conversions, Rust provides more ergonomic alternatives.

Coercions

Types can implicitly be coerced to change in certain contexts. These changes are generally just *weakening* of types, largely focused around pointers and lifetimes. They mostly exist to make Rust "just work" in more cases, and are largely harmless.

Here's all the kinds of coercion:

Coercion is allowed between the following types:

- Transitivity: `T_1` to `T_3` where `T_1` coerces to `T_2` and `T_2` coerces to `T_3`
- Pointer Weakening:
  - `&mut T` to `&T`
  - `*mut T` to `*const T`
  - `&T` to `*const T`
  - `&mut T` to `*mut T`
- Unsizing: `T` to `U` if `T` implements `CoerceUnsized<U>`

`CoerceUnsized<Pointer<U>> for Pointer<T> where T: Unsize<U>` is implemented for all pointer types (including smart pointers like Box and Rc). Unsize is only implemented automatically, and enables the following transformations:

- `[T; n]` => `[T]`
- `T` => `Trait` where `T: Trait`
- `Foo<..., T, ...>` => `Foo<..., U, ...>` where:
  - `T: Unsize<U>`
  - `Foo` is a struct
  - Only the last field of `Foo` has type `T`
  - `T` is not part of the type of any other fields

Coercions occur at a *coercion site*. Any location that is explicitly typed will cause a coercion to its type. If inference is necessary, the coercion will not be performed. Exhaustively, the coercion sites for an expression `e` to type `U` are:

- let statements, statics, and consts: `let x: U = e`
- Arguments to functions: `takes_a_U(e)`
- Any expression that will be returned: `fn foo() -> U { e }`
- Struct literals: `Foo { some_u: e }`
- Array literals: `let x: [U; 10] = [e, ..]`
- Tuple literals: `let x: (U, ..) = (e, ..)`
- The last expression in a block: `let x: U = { ..; e }`

Note that we do not perform coercions when matching traits (except for receivers, see below). If there is an impl for some type `U` and `T` coerces to `U`, that does not constitute an implementation for `T`. For example, the following will not type check, even though it is OK to coerce `t` to `&T` and there is an impl for `&T`:

```
trait Trait {}

fn foo<X: Trait>(t: X) {}

impl<'a> Trait for &'a i32 {}


fn main() {
    let t: &mut i32 = &mut 0;
    foo(t);
}
```

```
<anon>:10:5: 10:8 error: the trait `Trait` is not implemented for the type `&mut i32`
↳ [E0277]
<anon>:10     foo(t);
              ^~~
```

The Dot Operator

The dot operator will perform a lot of magic to convert types. It will perform auto-referencing, auto-dereferencing, and coercion until types match.

TODO: steal information from http://stackoverflow.com/questions/28519997/what-are-rusts-exact-auto-dereferencing-rules/28552082#28552082

Casts

Casts are a superset of coercions: every coercion can be explicitly invoked via a cast. However some conversions require a cast. While coercions are pervasive and largely harmless, these "true casts" are rare and potentially dangerous. As such, casts must be explicitly invoked using the `as` keyword: `expr as Type`.

True casts generally revolve around raw pointers and the primitive numeric types. Even though they're dangerous, these casts are infallible at runtime. If a cast triggers some subtle corner case no indication will be given that this occurred. The cast will simply succeed. That said, casts must be valid at the type level, or else they will be prevented statically. For instance, `7u8 as bool` will not compile.

That said, casts aren't `unsafe` because they generally can't violate memory safety *on their own*. For instance, converting an integer to a raw pointer can very easily lead to terrible things. However the act of creating the pointer itself is safe, because actually using a raw pointer is already marked as `unsafe`.

Here's an exhaustive list of all the true casts. For brevity, we will use * to denote either a `*const` or `*mut`, and `integer` to denote any integral primitive:

- `*T as *U` where `T, U: Sized`

- `*T as *U` TODO: explain unsized situation
- `*T as integer`
- `integer as *T`
- `number as number`
- `C-like-enum as integer`
- `bool as integer`
- `char as integer`
- `u8 as char`
- `&[T; n] as *const T`
- `fn as *T` where `T: Sized`
- `fn as integer`

Note that lengths are not adjusted when casting raw slices - `*const [u16] as *const [u8]` creates a slice that only includes half of the original memory.

Casting is not transitive, that is, even if `e as U1 as U2` is a valid expression, `e as U2` is not necessarily so.

For numeric casts, there are quite a few cases to consider:

- casting between two integers of the same size (e.g. i32 -> u32) is a no-op
- casting from a larger integer to a smaller integer (e.g. u32 -> u8) will truncate
- casting from a smaller integer to a larger integer (e.g. u8 -> u32) will
  - zero-extend if the source is unsigned
  - sign-extend if the source is signed
- casting from a float to an integer will round the float towards zero
  - **NOTE: currently this will cause Undefined Behaviour if the rounded value cannot be represented by the target integer type**[1]. This includes Inf and NaN. This is a bug and will be fixed.
- casting from an integer to float will produce the floating point representation of the integer, rounded if necessary (rounding strategy unspecified)
- casting from an f32 to an f64 is perfect and lossless
- casting from an f64 to an f32 will produce the closest possible value (rounding strategy unspecified)
  - **NOTE: currently this will cause Undefined Behaviour if the value is finite but larger or smaller than the largest or smallest finite value representable by f32**[2]. This is a bug and will be fixed.

Transmutes

Get out of our way type system! We're going to reinterpret these bits or die trying! Even though this book is all about doing things that are unsafe, I really can't emphasize that you should deeply think about finding Another Way than the operations covered in this section. This is

[1] https://github.com/rust-lang/rust/issues/10184
[2] https://github.com/rust-lang/rust/issues/15536

really, truly, the most horribly unsafe thing you can do in Rust. The railguards here are dental floss.

`mem::transmute<T, U>` takes a value of type `T` and reinterprets it to have type `U`. The only restriction is that the `T` and `U` are verified to have the same size. The ways to cause Undefined Behaviour with this are mind boggling.

- First and foremost, creating an instance of *any* type with an invalid state is going to cause arbitrary chaos that can't really be predicted.
- Transmute has an overloaded return type. If you do not specify the return type it may produce a surprising type to satisfy inference.
- Making a primitive with an invalid value is UB
- Transmuting between non-repr(C) types is UB
- Transmuting an & to &mut is UB
  - Transmuting an & to &mut is *always* UB
  - No you can't do it
  - No you're not special
- Transmuting to a reference without an explicitly provided lifetime produces an [unbounded lifetime]

`mem::transmute_copy<T, U>` somehow manages to be *even more* wildly unsafe than this. It copies `size_of<U>` bytes out of an `&T` and interprets them as a `U`. The size check that `mem::transmute` has is gone (as it may be valid to copy out a prefix), though it is Undefined Behaviour for `U` to be larger than `T`.

Also of course you can get most of the functionality of these functions using pointer casts.

6

Uninitialized Memory

All runtime-allocated memory in a Rust program begins its life as *uninitialized*. In this state the value of the memory is an indeterminate pile of bits that may or may not even reflect a valid state for the type that is supposed to inhabit that location of memory. Attempting to interpret this memory as a value of *any* type will cause Undefined Behaviour. Do Not Do This.

Rust provides mechanisms to work with uninitialized memory in checked (safe) and unchecked (unsafe) ways.

Checked

Like C, all stack variables in Rust are uninitialized until a value is explicitly assigned to them. Unlike C, Rust statically prevents you from ever reading them until you do:

```
fn main() {
    let x: i32;
    println!("{}", x);
}
```

```
src/main.rs:3:20: 3:21 error: use of possibly uninitialized variable: `x`
src/main.rs:3     println!("{}", x);
                                 ^
```

This is based off of a basic branch analysis: every branch must assign a value to `x` before it is first used. Interestingly, Rust doesn't require the variable to be mutable to perform a delayed initialization if every branch assigns exactly once. However the analysis does not take advantage of constant analysis or anything like that. So this compiles:

```
fn main() {
    let x: i32;

    if true {
        x = 1;
    } else {
        x = 2;
    }

    println!("{}", x);
}
```

but this doesn't:

```
fn main() {
    let x: i32;
    if true {
        x = 1;
    }
    println!("{}", x);
}
```

```
src/main.rs:6:17: 6:18 error: use of possibly uninitialized variable: `x`
src/main.rs:6   println!("{}", x);
```

while this does:

```
fn main() {
    let x: i32;
    if true {
        x = 1;
        println!("{}", x);
    }
    // Don't care that there are branches where it's not initialized
    // since we don't use the value in those branches
}
```

Of course, while the analysis doesn't consider actual values, it does have a relatively sophisticated understanding of dependencies and control flow. For instance, this works:

```
let x: i32;

loop {
    // Rust doesn't understand that this branch will be taken unconditionally,
    // because it relies on actual values.
    if true {
        // But it does understand that it will only be taken once because
```

```
        // we unconditionally break out of it. Therefore `x` doesn't
        // need to be marked as mutable.
        x = 0;
        break;
    }
}
// It also knows that it's impossible to get here without reaching the break.
// And therefore that `x` must be initialized here!
println!("{}", x);
```

If a value is moved out of a variable, that variable becomes logically uninitialized if the type of the value isn't Copy. That is:

```
fn main() {
    let x = 0;
    let y = Box::new(0);
    let z1 = x; // x is still valid because i32 is Copy
    let z2 = y; // y is now logically uninitialized because Box isn't Copy
}
```

However reassigning `y` in this example *would* require `y` to be marked as mutable, as a Safe Rust program could observe that the value of `y` changed:

```
fn main() {
    let mut y = Box::new(0);
    let z = y; // y is now logically uninitialized because Box isn't Copy
    y = Box::new(1); // reinitialize y
}
```

Otherwise it's like `y` is a brand new variable.

Drop Flags

The examples in the previous section introduce an interesting problem for Rust. We have seen that it's possible to conditionally initialize, deinitialize, and reinitialize locations of memory totally safely. For Copy types, this isn't particularly notable since they're just a random pile of bits. However types with destructors are a different story: Rust needs to know whether to call a destructor whenever a variable is assigned to, or a variable goes out of scope. How can it do this with conditional initialization?

Note that this is not a problem that all assignments need worry about. In particular, assigning through a dereference unconditionally drops, and assigning in a `let` unconditionally doesn't drop:

```
let mut x = Box::new(0); // let makes a fresh variable, so never need to drop
let y = &mut x;
*y = Box::new(1); // Deref assumes the referent is initialized, so always drops
```

This is only a problem when overwriting a previously initialized variable or one of its subfields.

It turns out that Rust actually tracks whether a type should be dropped or not *at runtime*. As a variable becomes initialized and uninitialized, a *drop flag* for that variable is toggled. When a variable might need to be dropped, this flag is evaluated to determine if it should be dropped.

Of course, it is often the case that a value's initialization state can be statically known at every point in the program. If this is the case, then the compiler can theoretically generate more efficient code! For instance, straight- line code has such *static drop semantics*:

```
let mut x = Box::new(0); // x was uninit; just overwrite.
let mut y = x;           // y was uninit; just overwrite and make x uninit.
x = Box::new(0);         // x was uninit; just overwrite.
y = x;                   // y was init; Drop y, overwrite it, and make x uninit!
                         // y goes out of scope; y was init; Drop y!
                         // x goes out of scope; x was uninit; do nothing.
```

Similarly, branched code where all branches have the same behaviour with respect to initialization has static drop semantics:

```
let mut x = Box::new(0);    // x was uninit; just overwrite.
if condition {
    drop(x)                 // x gets moved out; make x uninit.
} else {
    println!("{}", x);
    drop(x)                 // x gets moved out; make x uninit.
}
x = Box::new(0);            // x was uninit; just overwrite.
                            // x goes out of scope; x was init; Drop x!
```

However code like this *requires* runtime information to correctly Drop:

```
let x;
if condition {
    x = Box::new(0);        // x was uninit; just overwrite.
    println!("{}", x);
}
                            // x goes out of scope; x might be uninit;
                            // check the flag!
```

Of course, in this case it's trivial to retrieve static drop semantics:

```
if condition {
    let x = Box::new(0);
    println!("{}", x);
}
```

As of Rust 1.0, the drop flags are actually not-so-secretly stashed in a hidden field of any type that implements Drop. Rust sets the drop flag by overwriting the entire value with a particular bit

pattern. This is pretty obviously Not The Fastest and causes a bunch of trouble with optimizing code. It's legacy from a time when you could do much more complex conditional initialization.

As such work is currently under way to move the flags out onto the stack frame where they more reasonably belong. Unfortunately, this work will take some time as it requires fairly substantial changes to the compiler.

Regardless, Rust programs don't need to worry about uninitialized values on the stack for correctness. Although they might care for performance. Thankfully, Rust makes it easy to take control here! Uninitialized values are there, and you can work with them in Safe Rust, but you're never in danger.

Unchecked

One interesting exception to this rule is working with arrays. Safe Rust doesn't permit you to partially initialize an array. When you initialize an array, you can either set every value to the same thing with `let x = [val; N]`, or you can specify each member individually with `let x = [val1, val2, val3]`. Unfortunately this is pretty rigid, especially if you need to initialize your array in a more incremental or dynamic way.

Unsafe Rust gives us a powerful tool to handle this problem: `mem::uninitialized`. This function pretends to return a value when really it does nothing at all. Using it, we can convince Rust that we have initialized a variable, allowing us to do trickier things with conditional and incremental initialization.

Unfortunately, this opens us up to all kinds of problems. Assignment has a different meaning to Rust based on whether it believes that a variable is initialized or not. If it's believed uninitialized, then Rust will semantically just memcopy the bits over the uninitialized ones, and do nothing else. However if Rust believes a value to be initialized, it will try to `Drop` the old value! Since we've tricked Rust into believing that the value is initialized, we can no longer safely use normal assignment.

This is also a problem if you're working with a raw system allocator, which returns a pointer to uninitialized memory.

To handle this, we must use the `ptr` module. In particular, it provides three functions that allow us to assign bytes to a location in memory without dropping the old value: `write`, `copy`, and `copy_nonoverlapping`.

- `ptr::write(ptr, val)` takes a `val` and moves it into the address pointed to by `ptr`.
- `ptr::copy(src, dest, count)` copies the bits that `count` T's would occupy from src to dest. (this is equivalent to memmove – note that the argument order is reversed!)
- `ptr::copy_nonoverlapping(src, dest, count)` does what `copy` does, but a little faster on the assumption that the two ranges of memory don't overlap. (this is equivalent to memcpy – note that the argument order is reversed!)

It should go without saying that these functions, if misused, will cause serious havoc or just straight up Undefined Behaviour. The only things that these functions *themselves* require is that the locations you want to read and write are allocated. However the ways writing arbitrary bits to arbitrary locations of memory can break things are basically uncountable!

Putting this all together, we get the following:

```
use std::mem;
use std::ptr;

// size of the array is hard-coded but easy to change. This means we can't
// use [a, b, c] syntax to initialize the array, though!
const SIZE: usize = 10;

let mut x: [Box<u32>; SIZE];

unsafe {
    // convince Rust that x is Totally Initialized
    x = mem::uninitialized();
    for i in 0..SIZE {
        // very carefully overwrite each index without reading it
        // NOTE: exception safety is not a concern; Box can't panic
        ptr::write(&mut x[i], Box::new(i as u32));
    }
}

println!("{:?}", x);
```

It's worth noting that you don't need to worry about `ptr::write`-style shenanigans with types which don't implement `Drop` or contain `Drop` types, because Rust knows not to try to drop them. Similarly you should be able to assign to fields of partially initialized structs directly if those fields don't contain any `Drop` types.

However when working with uninitialized memory you need to be ever-vigilant for Rust trying to drop values you make like this before they're fully initialized. Every control path through that variable's scope must initialize the value before it ends, if it has a destructor. *This includes code panicking (chapter 8, page 73).*

And that's about it for working with uninitialized memory! Basically nothing anywhere expects to be handed uninitialized memory, so if you're going to pass it around at all, be sure to be *really* careful.

7

Ownership Based Resource Management

OBRM (AKA RAII: Resource Acquisition Is Initialization) is something you'll interact with a lot in Rust. Especially if you use the standard library.

Roughly speaking the pattern is as follows: to acquire a resource, you create an object that manages it. To release the resource, you simply destroy the object, and it cleans up the resource for you. The most common "resource" this pattern manages is simply *memory*. `Box`, `Rc`, and basically everything in `std::collections` is a convenience to enable correctly managing memory. This is particularly important in Rust because we have no pervasive GC to rely on for memory management. Which is the point, really: Rust is about control. However we are not limited to just memory. Pretty much every other system resource like a thread, file, or socket is exposed through this kind of API.

Constructors

There is exactly one way to create an instance of a user-defined type: name it, and initialize all its fields at once:

```
struct Foo {
    a: u8,
    b: u32,
    c: bool,
}

enum Bar {
    X(u32),
    Y(bool),
}

struct Unit;
```

```
let foo = Foo { a: 0, b: 1, c: false };
let bar = Bar::X(0);
let empty = Unit;
```

That's it. Every other way you make an instance of a type is just calling a totally vanilla function that does some stuff and eventually bottoms out to The One True Constructor.

Unlike C++, Rust does not come with a slew of built-in kinds of constructor. There are no Copy, Default, Assignment, Move, or whatever constructors. The reasons for this are varied, but it largely boils down to Rust's philosophy of *being explicit*.

Move constructors are meaningless in Rust because we don't enable types to "care" about their location in memory. Every type must be ready for it to be blindly memcopied to somewhere else in memory. This means pure on-the-stack-but- still-movable intrusive linked lists are simply not happening in Rust (safely).

Assignment and copy constructors similarly don't exist because move semantics are the only semantics in Rust. At most `x = y` just moves the bits of y into the x variable. Rust does provide two facilities for providing C++'s copy- oriented semantics: `Copy` and `Clone`. Clone is our moral equivalent of a copy constructor, but it's never implicitly invoked. You have to explicitly call `clone` on an element you want to be cloned. Copy is a special case of Clone where the implementation is just "copy the bits". Copy types *are* implicitly cloned whenever they're moved, but because of the definition of Copy this just means not treating the old copy as uninitialized – a no-op.

While Rust provides a `Default` trait for specifying the moral equivalent of a default constructor, it's incredibly rare for this trait to be used. This is because variables aren't implicitly initialized (chapter 6, page 57). Default is basically only useful for generic programming. In concrete contexts, a type will provide a static `new` method for any kind of "default" constructor. This has no relation to `new` in other languages and has no special meaning. It's just a naming convention.

TODO: talk about "placement new"?

Destructors

What the language *does* provide is full-blown automatic destructors through the `Drop` trait, which provides the following method:

```
fn drop(&mut self);
```

This method gives the type time to somehow finish what it was doing.

After `drop` is run, Rust will recursively try to drop all of the fields of `self`.

This is a convenience feature so that you don't have to write "destructor boilerplate" to drop children. If a struct has no special logic for being dropped other than dropping its children, then it means `Drop` doesn't need to be implemented at all!

There is no stable way to prevent this behaviour in Rust 1.0.

Note that taking `&mut self` means that even if you could suppress recursive Drop, Rust will prevent you from e.g. moving fields out of self. For most types, this is totally fine.

For instance, a custom implementation of `Box` might write `Drop` like this:

```
#![feature(alloc, heap_api, core_intrinsics, unique)]

extern crate alloc;

use std::ptr::Unique;
use std::intrinsics::drop_in_place;
use std::mem;

use alloc::heap;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(*self.ptr);
            heap::deallocate((*self.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}
```

and this works fine because when Rust goes to drop the `ptr` field it just sees a [Unique] that has no actual `Drop` implementation. Similarly nothing can use-after-free the `ptr` because when drop exits, it becomes inacessible.

However this wouldn't work:

```
#![feature(alloc, heap_api, core_intrinsics, unique)]

extern crate alloc;

use std::ptr::Unique;
use std::intrinsics::drop_in_place;
use std::mem;

use alloc::heap;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(*self.ptr);
```

```
            heap::deallocate((*self.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Box<T> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // Hyper-optimized: deallocate the box's contents for it
            // without `drop`ing the contents
            heap::deallocate((*self.my_box.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}
```

After we deallocate the `box`'s ptr in SuperBox's destructor, Rust will happily proceed to tell the box to Drop itself and everything will blow up with use-after-frees and double-frees.

Note that the recursive drop behaviour applies to all structs and enums regardless of whether they implement Drop. Therefore something like

```
struct Boxy<T> {
    data1: Box<T>,
    data2: Box<T>,
    info: u32,
}
```

will have its data1 and data2's fields destructors whenever it "would" be dropped, even though it itself doesn't implement Drop. We say that such a type *needs Drop*, even though it is not itself Drop.

Similarly,

```
enum Link {
    Next(Box<Link>),
    None,
}
```

will have its inner Box field dropped if and only if an instance stores the Next variant.

In general this works really nicely because you don't need to worry about adding/removing drops when you refactor your data layout. Still there's certainly many valid usecases for needing to do trickier things with destructors.

The classic safe solution to overriding recursive drop and allowing moving out of Self during `drop` is to use an Option:

```
#![feature(alloc, heap_api, core_intrinsics, unique)]

extern crate alloc;

use std::ptr::Unique;
use std::intrinsics::drop_in_place;
use std::mem;

use alloc::heap;

struct Box<T>{ ptr: Unique<T> }

impl<T> Drop for Box<T> {
    fn drop(&mut self) {
        unsafe {
            drop_in_place(*self.ptr);
            heap::deallocate((*self.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
        }
    }
}

struct SuperBox<T> { my_box: Option<Box<T>> }

impl<T> Drop for SuperBox<T> {
    fn drop(&mut self) {
        unsafe {
            // Hyper-optimized: deallocate the box's contents for it
            // without `drop`ing the contents. Need to set the `box`
            // field as `None` to prevent Rust from trying to Drop it.
            let my_box = self.my_box.take().unwrap();
            heap::deallocate((*my_box.ptr) as *mut u8,
                             mem::size_of::<T>(),
                             mem::align_of::<T>());
            mem::forget(my_box);
        }
    }
}
```

However this has fairly odd semantics: you're saying that a field that *should* always be Some *may* be None, just because that happens in the destructor. Of course this conversely makes a lot of sense: you can call arbitrary methods on self during the destructor, and this should prevent you from ever doing so after deinitializing the field. Not that it will prevent you from producing any other arbitrarily invalid state in there.

On balance this is an ok choice. Certainly what you should reach for by default. However, in the

future we expect there to be a first-class way to announce that a field shouldn't be automatically dropped.

Leaking

Ownership-based resource management is intended to simplify composition. You acquire resources when you create the object, and you release the resources when it gets destroyed. Since destruction is handled for you, it means you can't forget to release the resources, and it happens as soon as possible! Surely this is perfect and all of our problems are solved.

Everything is terrible and we have new and exotic problems to try to solve.

Many people like to believe that Rust eliminates resource leaks. In practice, this is basically true. You would be surprised to see a Safe Rust program leak resources in an uncontrolled way.

However from a theoretical perspective this is absolutely not the case, no matter how you look at it. In the strictest sense, "leaking" is so abstract as to be unpreventable. It's quite trivial to initialize a collection at the start of a program, fill it with tons of objects with destructors, and then enter an infinite event loop that never refers to it. The collection will sit around uselessly, holding on to its precious resources until the program terminates (at which point all those resources would have been reclaimed by the OS anyway).

We may consider a more restricted form of leak: failing to drop a value that is unreachable. Rust also doesn't prevent this. In fact Rust *has a function for doing this*: `mem::forget`. This function consumes the value it is passed *and then doesn't run its destructor*.

In the past `mem::forget` was marked as unsafe as a sort of lint against using it, since failing to call a destructor is generally not a well-behaved thing to do (though useful for some special unsafe code). However this was generally determined to be an untenable stance to take: there are many ways to fail to call a destructor in safe code. The most famous example is creating a cycle of reference-counted pointers using interior mutability.

It is reasonable for safe code to assume that destructor leaks do not happen, as any program that leaks destructors is probably wrong. However *unsafe* code cannot rely on destructors to be run in order to be safe. For most types this doesn't matter: if you leak the destructor then the type is by definition inaccessible, so it doesn't matter, right? For instance, if you leak a `Box<u8>` then you waste some memory but that's hardly going to violate memory-safety.

However where we must be careful with destructor leaks are *proxy* types. These are types which manage access to a distinct object, but don't actually own it. Proxy objects are quite rare. Proxy objects you'll need to care about are even rarer. However we'll focus on three interesting examples in the standard library:

- `vec::Drain`
- `Rc`
- `thread::scoped::JoinGuard`

Drain

`drain` is a collections API that moves data out of the container without consuming the container. This enables us to reuse the allocation of a `Vec` after claiming ownership over all of its contents.

It produces an iterator (Drain) that returns the contents of the Vec by-value.

Now, consider Drain in the middle of iteration: some values have been moved out, and others haven't. This means that part of the Vec is now full of logically uninitialized data! We could backshift all the elements in the Vec every time we remove a value, but this would have pretty catastrophic performance consequences.

Instead, we would like Drain to fix the Vec's backing storage when it is dropped. It should run itself to completion, backshift any elements that weren't removed (drain supports subranges), and then fix Vec's `len`. It's even unwinding-safe! Easy!

Now consider the following:

```
let mut vec = vec![Box::new(0); 4];

{
    // start draining, vec can no longer be accessed
    let mut drainer = vec.drain(..);

    // pull out two elements and immediately drop them
    drainer.next();
    drainer.next();

    // get rid of drainer, but don't call its destructor
    mem::forget(drainer);
}

// Oops, vec[0] was dropped, we're reading a pointer into free'd memory!
println!("{}", vec[0]);
```

This is pretty clearly Not Good. Unfortunately, we're kind've stuck between a rock and a hard place: maintaining consistent state at every step has an enormous cost (and would negate any benefits of the API). Failing to maintain consistent state gives us Undefined Behaviour in safe code (making the API unsound).

So what can we do? Well, we can pick a trivially consistent state: set the Vec's len to be 0 when we start the iteration, and fix it up if necessary in the destructor. That way, if everything executes like normal we get the desired behaviour with minimal overhead. But if someone has the *audacity* to mem::forget us in the middle of the iteration, all that does is *leak even more* (and possibly leave the Vec in an unexpected but otherwise consistent state). Since we've accepted that mem::forget is safe, this is definitely safe. We call leaks causing more leaks a *leak amplification*.

Rc

Rc is an interesting case because at first glance it doesn't appear to be a proxy value at all. After all, it manages the data it points to, and dropping all the Rcs for a value will drop that value. Leaking an Rc doesn't seem like it would be particularly dangerous. It will leave the refcount permanently incremented and prevent the data from being freed or dropped, but that seems just like Box, right?

Nope.

Let's consider a simplified implementation of Rc:

```
struct Rc<T> {
    ptr: *mut RcBox<T>,
}

struct RcBox<T> {
    data: T,
    ref_count: usize,
}

impl<T> Rc<T> {
    fn new(data: T) -> Self {
        unsafe {
            // Wouldn't it be nice if heap::allocate worked like this?
            let ptr = heap::allocate<RcBox<T>>();
            ptr::write(ptr, RcBox {
                data: data,
                ref_count: 1,
            });
            Rc { ptr: ptr }
        }
    }

    fn clone(&self) -> Self {
        unsafe {
            (*self.ptr).ref_count += 1;
        }
        Rc { ptr: self.ptr }
    }
}

impl<T> Drop for Rc<T> {
    fn drop(&mut self) {
        unsafe {
            (*self.ptr).ref_count -= 1;
            if (*self.ptr).ref_count == 0 {
                // drop the data and then free it
                ptr::read(self.ptr);
                heap::deallocate(self.ptr);
            }
        }
    }
}
```

This code contains an implicit and subtle assumption: `ref_count` can fit in a `usize`, because there can't be more than `usize::MAX` Rcs in memory. However this itself assumes that the `ref_count` accurately reflects the number of Rcs in memory, which we know is false with `mem::forget`. Using `mem::forget` we can overflow the `ref_count`, and then get it down to 0 with outstanding Rcs.

Then we can happily use-after-free the inner data. Bad Bad Not Good.

This can be solved by just checking the `ref_count` and doing *something*. The standard library's stance is to just abort, because your program has become horribly degenerate. Also *oh my gosh* it's such a ridiculous corner case.

thread::scoped::JoinGuard

The thread::scoped API intends to allow threads to be spawned that reference data on their parent's stack without any synchronization over that data by ensuring the parent joins the thread before any of the shared data goes out of scope.

```
pub fn scoped<'a, F>(f: F) -> JoinGuard<'a>
    where F: FnOnce() + Send + 'a
```

Here `f` is some closure for the other thread to execute. Saying that `F: Send +'a` is saying that it closes over data that lives for `'a`, and it either owns that data or the data was Sync (implying `&data` is Send).

Because JoinGuard has a lifetime, it keeps all the data it closes over borrowed in the parent thread. This means the JoinGuard can't outlive the data that the other thread is working on. When the JoinGuard *does* get dropped it blocks the parent thread, ensuring the child terminates before any of the closed-over data goes out of scope in the parent.

Usage looked like:

```
let mut data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
{
    let guards = vec![];
    for x in &mut data {
        // Move the mutable reference into the closure, and execute
        // it on a different thread. The closure has a lifetime bound
        // by the lifetime of the mutable reference `x` we store in it.
        // The guard that is returned is in turn assigned the lifetime
        // of the closure, so it also mutably borrows `data` as `x` did.
        // This means we cannot access `data` until the guard goes away.
        let guard = thread::scoped(move || {
            *x *= 2;
        });
        // store the thread's guard for later
        guards.push(guard);
    }
    // All guards are dropped here, forcing the threads to join
    // (this thread blocks here until the others terminate).
    // Once the threads join, the borrow expires and the data becomes
    // accessible again in this thread.
}
// data is definitely mutated here.
```

In principle, this totally works! Rust's ownership system perfectly ensures it! ...except it relies on a destructor being called to be safe.

```
let mut data = Box::new(0);
{
    let guard = thread::scoped(|| {
        // This is at best a data race. At worst, it's also a use-after-free.
        *data += 1;
    });
    // Because the guard is forgotten, expiring the loan without blocking this
    // thread.
    mem::forget(guard);
}
// So the Box is dropped here while the scoped thread may or may not be trying
// to access it.
```

Dang. Here the destructor running was pretty fundamental to the API, and it had to be scrapped in favour of a completely different design.

8

Unwinding

Rust has a *tiered* error-handling scheme:

- If something might reasonably be absent, Option is used.
- If something goes wrong and can reasonably be handled, Result is used.
- If something goes wrong and cannot reasonably be handled, the thread panics.
- If something catastrophic happens, the program aborts.

Option and Result are overwhelmingly preferred in most situations, especially since they can be promoted into a panic or abort at the API user's discretion. Panics cause the thread to halt normal execution and unwind its stack, calling destructors as if every function instantly returned.

As of 1.0, Rust is of two minds when it comes to panics. In the long-long-ago, Rust was much more like Erlang. Like Erlang, Rust had lightweight tasks, and tasks were intended to kill themselves with a panic when they reached an untenable state. Unlike an exception in Java or C++, a panic could not be caught at any time. Panics could only be caught by the owner of the task, at which point they had to be handled or *that* task would itself panic.

Unwinding was important to this story because if a task's destructors weren't called, it would cause memory and other system resources to leak. Since tasks were expected to die during normal execution, this would make Rust very poor for long-running systems!

As the Rust we know today came to be, this style of programming grew out of fashion in the push for less-and-less abstraction. Light-weight tasks were killed in the name of heavy-weight OS threads. Still, on stable Rust as of 1.0 panics can only be caught by the parent thread. This means catching a panic requires spinning up an entire OS thread! This unfortunately stands in conflict to Rust's philosophy of zero-cost abstractions.

There is an unstable API called `catch_panic` that enables catching a panic without spawning a thread. Still, we would encourage you to only do this sparingly. In particular, Rust's current unwinding implementation is heavily optimized for the "doesn't unwind" case. If a program doesn't unwind, there should be no runtime cost for the program being *ready* to unwind. As a consequence, actually unwinding will be more expensive than in e.g. Java. Don't build your

programs to unwind under normal circumstances. Ideally, you should only panic for programming errors or *extreme* problems.

Rust's unwinding strategy is not specified to be fundamentally compatible with any other language's unwinding. As such, unwinding into Rust from another language, or unwinding into another language from Rust is Undefined Behaviour. You must *absolutely* catch any panics at the FFI boundary! What you do at that point is up to you, but *something* must be done. If you fail to do this, at best, your application will crash and burn. At worst, your application *won't* crash and burn, and will proceed with completely clobbered state.

Exception Safety

Although programs should use unwinding sparingly, there's a lot of code that *can* panic. If you unwrap a None, index out of bounds, or divide by 0, your program will panic. On debug builds, every arithmetic operation can panic if it overflows. Unless you are very careful and tightly control what code runs, pretty much everything can unwind, and you need to be ready for it.

Being ready for unwinding is often referred to as *exception safety* in the broader programming world. In Rust, there are two levels of exception safety that one may concern themselves with:

- In unsafe code, we *must* be exception safe to the point of not violating memory safety. We'll call this *minimal* exception safety.
- In safe code, it is *good* to be exception safe to the point of your program doing the right thing. We'll call this *maximal* exception safety.

As is the case in many places in Rust, Unsafe code must be ready to deal with bad Safe code when it comes to unwinding. Code that transiently creates unsound states must be careful that a panic does not cause that state to be used. Generally this means ensuring that only non-panicking code is run while these states exist, or making a guard that cleans up the state in the case of a panic. This does not necessarily mean that the state a panic witnesses is a fully coherent state. We need only guarantee that it's a *safe* state.

Most Unsafe code is leaf-like, and therefore fairly easy to make exception-safe. It controls all the code that runs, and most of that code can't panic. However it is not uncommon for Unsafe code to work with arrays of temporarily uninitialized data while repeatedly invoking caller-provided code. Such code needs to be careful and consider exception safety.

Vec::push_all

`Vec::push_all` is a temporary hack to get extending a Vec by a slice reliably efficient without specialization. Here's a simple implementation:

```
impl<T: Clone> Vec<T> {
    fn push_all(&mut self, to_push: &[T]) {
        self.reserve(to_push.len());
        unsafe {
            // can't overflow because we just reserved this
```

```
            self.set_len(self.len() + to_push.len());

            for (i, x) in to_push.iter().enumerate() {
                self.ptr().offset(i as isize).write(x.clone());
            }
        }
    }
}
```

We bypass `push` in order to avoid redundant capacity and `len` checks on the Vec that we definitely know has capacity. The logic is totally correct, except there's a subtle problem with our code: it's not exception-safe! `set_len`, `offset`, and `write` are all fine; `clone` is the panic bomb we over-looked.

Clone is completely out of our control, and is totally free to panic. If it does, our function will exit early with the length of the Vec set too large. If the Vec is looked at or dropped, uninitialized memory will be read!

The fix in this case is fairly simple. If we want to guarantee that the values we *did* clone are dropped, we can set the `len` every loop iteration. If we just want to guarantee that uninitialized memory can't be observed, we can set the `len` after the loop.

BinaryHeap::sift_up

Bubbling an element up a heap is a bit more complicated than extending a Vec. The pseudocode is as follows:

```
bubble_up(heap, index):
    while index != 0 && heap[index] < heap[parent(index)]:
        heap.swap(index, parent(index))
        index = parent(index)
```

A literal transcription of this code to Rust is totally fine, but has an annoying performance characteristic: the `self` element is swapped over and over again uselessly. We would rather have the following:

```
bubble_up(heap, index):
    let elem = heap[index]
    while index != 0 && element < heap[parent(index)]:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

This code ensures that each element is copied as little as possible (it is in fact necessary that elem be copied twice in general). However it now exposes some exception safety trouble! At all times, there exists two copies of one value. If we panic in this function something will be double-dropped. Unfortunately, we also don't have full control of the code: that comparison is user-defined!

Unlike Vec, the fix isn't as easy here. One option is to break the user-defined code and the unsafe code into two separate phases:

```
bubble_up(heap, index):
    let end_index = index;
    while end_index != 0 && heap[end_index] < heap[parent(end_index)]:
        end_index = parent(end_index)

    let elem = heap[index]
    while index != end_index:
        heap[index] = heap[parent(index)]
        index = parent(index)
    heap[index] = elem
```

If the user-defined code blows up, that's no problem anymore, because we haven't actually touched the state of the heap yet. Once we do start messing with the heap, we're working with only data and functions that we trust, so there's no concern of panics.

Perhaps you're not happy with this design. Surely it's cheating! And we have to do the complex heap traversal *twice*! Alright, let's bite the bullet. Let's intermix untrusted and unsafe code *for reals*.

If Rust had `try` and `finally` like in Java, we could do the following:

```
bubble_up(heap, index):
    let elem = heap[index]
    try:
        while index != 0 && element < heap[parent(index)]:
            heap[index] = heap[parent(index)]
            index = parent(index)
    finally:
        heap[index] = elem
```

The basic idea is simple: if the comparison panics, we just toss the loose element in the logically uninitialized index and bail out. Anyone who observes the heap will see a potentially *inconsistent* heap, but at least it won't cause any double-drops! If the algorithm terminates normally, then this operation happens to coincide precisely with the how we finish up regardless.

Sadly, Rust has no such construct, so we're going to need to roll our own! The way to do this is to store the algorithm's state in a separate struct with a destructor for the "finally" logic. Whether we panic or not, that destructor will run and clean up after us.

```
struct Hole<'a, T: 'a> {
    data: &'a mut [T],
    /// `elt` is always `Some` from new until drop.
    elt: Option<T>,
    pos: usize,
}
```

```
impl<'a, T> Hole<'a, T> {
    fn new(data: &'a mut [T], pos: usize) -> Self {
        unsafe {
            let elt = ptr::read(&data[pos]);
            Hole {
                data: data,
                elt: Some(elt),
                pos: pos,
            }
        }
    }

    fn pos(&self) -> usize { self.pos }

    fn removed(&self) -> &T { self.elt.as_ref().unwrap() }

    unsafe fn get(&self, index: usize) -> &T { &self.data[index] }

    unsafe fn move_to(&mut self, index: usize) {
        let index_ptr: *const _ = &self.data[index];
        let hole_ptr = &mut self.data[self.pos];
        ptr::copy_nonoverlapping(index_ptr, hole_ptr, 1);
        self.pos = index;
    }
}

impl<'a, T> Drop for Hole<'a, T> {
    fn drop(&mut self) {
        // fill the hole again
        unsafe {
            let pos = self.pos;
            ptr::write(&mut self.data[pos], self.elt.take().unwrap());
        }
    }
}

impl<T: Ord> BinaryHeap<T> {
    fn sift_up(&mut self, pos: usize) {
        unsafe {
            // Take out the value at `pos` and create a hole.
            let mut hole = Hole::new(&mut self.data, pos);

            while hole.pos() != 0 {
                let parent = parent(hole.pos());
                if hole.removed() <= hole.get(parent) { break }
                hole.move_to(parent);
            }
            // Hole will be unconditionally filled here; panic or not!
```

```
        }
    }
}
```

Poisoning

Although all unsafe code *must* ensure it has minimal exception safety, not all types ensure *maximal* exception safety. Even if the type does, your code may ascribe additional meaning to it. For instance, an integer is certainly exception-safe, but has no semantics on its own. It's possible that code that panics could fail to correctly update the integer, producing an inconsistent program state.

This is *usually* fine, because anything that witnesses an exception is about to get destroyed. For instance, if you send a Vec to another thread and that thread panics, it doesn't matter if the Vec is in a weird state. It will be dropped and go away forever. However some types are especially good at smuggling values across the panic boundary.

These types may choose to explicitly *poison* themselves if they witness a panic. Poisoning doesn't entail anything in particular. Generally it just means preventing normal usage from proceeding. The most notable example of this is the standard library's Mutex type. A Mutex will poison itself if one of its MutexGuards (the thing it returns when a lock is obtained) is dropped during a panic. Any future attempts to lock the Mutex will return an `Err` or panic.

Mutex poisons not for true safety in the sense that Rust normally cares about. It poisons as a safety-guard against blindly using the data that comes out of a Mutex that has witnessed a panic while locked. The data in such a Mutex was likely in the middle of being modified, and as such may be in an inconsistent or incomplete state. It is important to note that one cannot violate memory safety with such a type if it is correctly written. After all, it must be minimally exception-safe!

However if the Mutex contained, say, a BinaryHeap that does not actually have the heap property, it's unlikely that any code that uses it will do what the author intended. As such, the program should not proceed normally. Still, if you're double-plus-sure that you can do *something* with the value, the Mutex exposes a method to get the lock anyway. It *is* safe, after all. Just maybe nonsense.

9

Concurrency

Rust as a language doesn't *really* have an opinion on how to do concurrency or parallelism. The standard library exposes OS threads and blocking sys-calls because everyone has those, and they're uniform enough that you can provide an abstraction over them in a relatively uncontroversial way. Message passing, green threads, and async APIs are all diverse enough that any abstraction over them tends to involve trade-offs that we weren't willing to commit to for 1.0.

However the way Rust models concurrency makes it relatively easy to design your own concurrency paradigm as a library and have everyone else's code Just Work with yours. Just require the right lifetimes and Send and Sync where appropriate and you're off to the races. Or rather, off to the… not… having… races.

Races

Safe Rust guarantees an absence of data races, which are defined as:

- two or more threads concurrently accessing a location of memory
- one of them is a write
- one of them is unsynchronized

A data race has Undefined Behaviour, and is therefore impossible to perform in Safe Rust. Data races are *mostly* prevented through rust's ownership system: it's impossible to alias a mutable reference, so it's impossible to perform a data race. Interior mutability makes this more complicated, which is largely why we have the Send and Sync traits (see below).

However Rust does not prevent general race conditions.

This is pretty fundamentally impossible, and probably honestly undesirable. Your hardware is racy, your OS is racy, the other programs on your computer are racy, and the world this all runs in is racy. Any system that could genuinely claim to prevent *all* race conditions would be pretty awful to use, if not just incorrect.

So it's perfectly "fine" for a Safe Rust program to get deadlocked or do something incredibly stupid with incorrect synchronization. Obviously such a program isn't very good, but Rust can only hold your hand so far. Still, a race condition can't violate memory safety in a Rust program on its own. Only in conjunction with some other unsafe code can a race condition actually violate memory safety. For instance:

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
// Arc so that the memory the AtomicUsize is stored in still exists for
// the other thread to increment, even if we completely finish executing
// before it. Rust won't compile the program without it, because of the
// lifetime requirements of thread::spawn!
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// `move` captures other_idx by-value, moving it into this thread
thread::spawn(move || {
    // It's ok to mutate idx because this value
    // is an atomic, so it can't cause a Data Race.
    other_idx.fetch_add(10, Ordering::SeqCst);
});

// Index with the value loaded from the atomic. This is safe because we
// read the atomic memory only once, and then pass a copy of that value
// to the Vec's indexing implementation. This indexing will be correctly
// bounds checked, and there's no chance of the value getting changed
// in the middle. However our program may panic if the thread we spawned
// managed to increment before this ran. A race condition because correct
// program execution (panicing is rarely correct) depends on order of
// thread execution.
println!("{}", data[idx.load(Ordering::SeqCst)]);
```

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];

let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

// `move` captures other_idx by-value, moving it into this thread
thread::spawn(move || {
    // It's ok to mutate idx because this value
    // is an atomic, so it can't cause a Data Race.
    other_idx.fetch_add(10, Ordering::SeqCst);
```

```
});

if idx.load(Ordering::SeqCst) < data.len() {
    unsafe {
        // Incorrectly loading the idx after we did the bounds check.
        // It could have changed. This is a race condition, *and dangerous*
        // because we decided to do `get_unchecked`, which is `unsafe`.
        println!("{}", data.get_unchecked(idx.load(Ordering::SeqCst)));
    }
}
```

Send and Sync

Not everything obeys inherited mutability, though. Some types allow you to multiply alias a location in memory while mutating it. Unless these types use synchronization to manage this access, they are absolutely not thread safe. Rust captures this through the `Send` and `Sync` traits.

- A type is Send if it is safe to send it to another thread.
- A type is Sync if it is safe to share between threads (`&T` is Send).

Send and Sync are fundamental to Rust's concurrency story. As such, a substantial amount of special tooling exists to make them work right. First and foremost, they're [unsafe traits]. This means that they are unsafe to implement, and other unsafe code can assume that they are correctly implemented. Since they're *marker traits* (they have no associated items like methods), correctly implemented simply means that they have the intrinsic properties an implementor should have. Incorrectly implementing Send or Sync can cause Undefined Behaviour.

Send and Sync are also automatically derived traits. This means that, unlike every other trait, if a type is composed entirely of Send or Sync types, then it is Send or Sync. Almost all primitives are Send and Sync, and as a consequence pretty much all types you'll ever interact with are Send and Sync.

Major exceptions include:

- raw pointers are neither Send nor Sync (because they have no safety guards).
- `UnsafeCell` isn't Sync (and therefore `Cell` and `RefCell` aren't).
- `Rc` isn't Send or Sync (because the refcount is shared and unsynchronized).

`Rc` and `UnsafeCell` are very fundamentally not thread-safe: they enable unsynchronized shared mutable state. However raw pointers are, strictly speaking, marked as thread-unsafe as more of a *lint*. Doing anything useful with a raw pointer requires dereferencing it, which is already unsafe. In that sense, one could argue that it would be "fine" for them to be marked as thread safe.

However it's important that they aren't thread safe to prevent types that contain them from being automatically marked as thread safe. These types have non-trivial untracked ownership, and it's unlikely that their author was necessarily thinking hard about thread safety. In the case of Rc, we have a nice example of a type that contains a `*mut` that is definitely not thread safe.

Types that aren't automatically derived can simply implement them if desired:

```
struct MyBox(*mut u8);

unsafe impl Send for MyBox {}
unsafe impl Sync for MyBox {}
```

In the *incredibly rare* case that a type is inappropriately automatically derived to be Send or Sync, then one can also unimplement Send and Sync:

```
#![feature(optin_builtin_traits)]

// I have some magic semantics for some synchronization primitive!
struct SpecialThreadToken(u8);

impl !Send for SpecialThreadToken {}
impl !Sync for SpecialThreadToken {}
```

Note that *in and of itself* it is impossible to incorrectly derive Send and Sync. Only types that are ascribed special meaning by other unsafe code can possible cause trouble by being incorrectly Send or Sync.

Most uses of raw pointers should be encapsulated behind a sufficient abstraction that Send and Sync can be derived. For instance all of Rust's standard collections are Send and Sync (when they contain Send and Sync types) in spite of their pervasive use of raw pointers to manage allocations and complex ownership. Similarly, most iterators into these collections are Send and Sync because they largely behave like an `&` or `&mut` into the collection.

TODO: better explain what can or can't be Send or Sync. Sufficient to appeal only to data races?

Atomics

Rust pretty blatantly just inherits C11's memory model for atomics. This is not due this model being particularly excellent or easy to understand. Indeed, this model is quite complex and known to have several flaws[1]. Rather, it is a pragmatic concession to the fact that *everyone* is pretty bad at modeling atomics. At very least, we can benefit from existing tooling and research around C.

Trying to fully explain the model in this book is fairly hopeless. It's defined in terms of madness-inducing causality graphs that require a full book to properly understand in a practical way. If you want all the nitty-gritty details, you should check out C's specification (Section 7.17)[2]. Still, we'll try to cover the basics and some of the problems Rust developers face.

The C11 memory model is fundamentally about trying to bridge the gap between the semantics we want, the optimizations compilers want, and the inconsistent chaos our hardware wants. *We* would like to just write programs and have them do exactly what we said but, you know, fast. Wouldn't that be great?

[1] http://plv.mpi-sws.org/c11comp/popl15.pdf
[2] http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899

Compiler Reordering

Compilers fundamentally want to be able to do all sorts of crazy transformations to reduce data dependencies and eliminate dead code. In particular, they may radically change the actual order of events, or make events never occur! If we write something like

```
x = 1;
y = 3;
x = 2;
```

The compiler may conclude that it would be best if your program did

```
x = 2;
y = 3;
```

This has inverted the order of events and completely eliminated one event. From a single-threaded perspective this is completely unobservable: after all the statements have executed we are in exactly the same state. But if our program is multi-threaded, we may have been relying on `x` to actually be assigned to 1 before `y` was assigned. We would like the compiler to be able to make these kinds of optimizations, because they can seriously improve performance. On the other hand, we'd also like to be able to depend on our program *doing the thing we said*.

Hardware Reordering

On the other hand, even if the compiler totally understood what we wanted and respected our wishes, our hardware might instead get us in trouble. Trouble comes from CPUs in the form of memory hierarchies. There is indeed a global shared memory space somewhere in your hardware, but from the perspective of each CPU core it is *so very far away* and *so very slow*. Each CPU would rather work with its local cache of the data and only go through all the anguish of talking to shared memory only when it doesn't actually have that memory in cache.

After all, that's the whole point of the cache, right? If every read from the cache had to run back to shared memory to double check that it hadn't changed, what would the point be? The end result is that the hardware doesn't guarantee that events that occur in the same order on *one* thread, occur in the same order on *another* thread. To guarantee this, we must issue special instructions to the CPU telling it to be a bit less smart.

For instance, say we convince the compiler to emit this logic:

```
initial state: x = 0, y = 1

THREAD 1        THREAD2
y = 3;          if x == 1 {
x = 1;              y *= 2;
                }
```

Ideally this program has 2 possible final states:

- `y = 3`: (thread 2 did the check before thread 1 completed)
- `y = 6`: (thread 2 did the check after thread 1 completed)

However there's a third potential state that the hardware enables:

- `y = 2`: (thread 2 saw `x = 1`, but not `y = 3`, and then overwrote `y = 3`)

It's worth noting that different kinds of CPU provide different guarantees. It is common to separate hardware into two categories: strongly-ordered and weakly- ordered. Most notably x86/64 provides strong ordering guarantees, while ARM provides weak ordering guarantees. This has two consequences for concurrent programming:

- Asking for stronger guarantees on strongly-ordered hardware may be cheap or even free because they already provide strong guarantees unconditionally. Weaker guarantees may only yield performance wins on weakly-ordered hardware.
- Asking for guarantees that are too weak on strongly-ordered hardware is more likely to *happen* to work, even though your program is strictly incorrect. If possible, concurrent algorithms should be tested on weakly-ordered hardware.

Data Accesses

The C11 memory model attempts to bridge the gap by allowing us to talk about the *causality* of our program. Generally, this is by establishing a *happens before* relationship between parts of the program and the threads that are running them. This gives the hardware and compiler room to optimize the program more aggressively where a strict happens-before relationship isn't established, but forces them to be more careful where one is established. The way we communicate these relationships are through *data accesses* and *atomic accesses*.

Data accesses are the bread-and-butter of the programming world. They are fundamentally unsynchronized and compilers are free to aggressively optimize them. In particular, data accesses are free to be reordered by the compiler on the assumption that the program is single-threaded. The hardware is also free to propagate the changes made in data accesses to other threads as lazily and inconsistently as it wants. Most critically, data accesses are how data races happen. Data accesses are very friendly to the hardware and compiler, but as we've seen they offer *awful* semantics to try to write synchronized code with. Actually, that's too weak.

It is literally impossible to write correct synchronized code using only data accesses.

Atomic accesses are how we tell the hardware and compiler that our program is multi-threaded. Each atomic access can be marked with an *ordering* that specifies what kind of relationship it establishes with other accesses. In practice, this boils down to telling the compiler and hardware certain things they *can't* do. For the compiler, this largely revolves around re-ordering of instructions. For the hardware, this largely revolves around how writes are propagated to other threads. The set of orderings Rust exposes are:

- Sequentially Consistent (SeqCst)
- Release
- Acquire

- Relaxed

(Note: We explicitly do not expose the C11 *consume* ordering)

TODO: negative reasoning vs positive reasoning? TODO: "can't forget to synchronize"

Sequentially Consistent

Sequentially Consistent is the most powerful of all, implying the restrictions of all other orderings. Intuitively, a sequentially consistent operation cannot be reordered: all accesses on one thread that happen before and after a SeqCst access stay before and after it. A data-race-free program that uses only sequentially consistent atomics and data accesses has the very nice property that there is a single global execution of the program's instructions that all threads agree on. This execution is also particularly nice to reason about: it's just an interleaving of each thread's individual executions. This does not hold if you start using the weaker atomic orderings.

The relative developer-friendliness of sequential consistency doesn't come for free. Even on strongly-ordered platforms sequential consistency involves emitting memory fences.

In practice, sequential consistency is rarely necessary for program correctness. However sequential consistency is definitely the right choice if you're not confident about the other memory orders. Having your program run a bit slower than it needs to is certainly better than it running incorrectly! It's also mechanically trivial to downgrade atomic operations to have a weaker consistency later on. Just change `SeqCst` to `Relaxed` and you're done! Of course, proving that this transformation is *correct* is a whole other matter.

Acquire-Release

Acquire and Release are largely intended to be paired. Their names hint at their use case: they're perfectly suited for acquiring and releasing locks, and ensuring that critical sections don't overlap.

Intuitively, an acquire access ensures that every access after it stays after it. However operations that occur before an acquire are free to be reordered to occur after it. Similarly, a release access ensures that every access before it stays before it. However operations that occur after a release are free to be reordered to occur before it.

When thread A releases a location in memory and then thread B subsequently acquires *the same* location in memory, causality is established. Every write that happened before A's release will be observed by B after its release. However no causality is established with any other threads. Similarly, no causality is established if A and B access *different* locations in memory.

Basic use of release-acquire is therefore simple: you acquire a location of memory to begin the critical section, and then release that location to end it. For instance, a simple spinlock might look like:

```
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
```

```
    let lock = Arc::new(AtomicBool::new(false)); // value answers "am I locked?"

    // ... distribute lock to threads somehow ...

    // Try to acquire the lock by setting it to true
    while lock.compare_and_swap(false, true, Ordering::Acquire) { }
    // broke out of the loop, so we successfully acquired the lock!

    // ... scary data accesses ...

    // ok we're done, release the lock
    lock.store(false, Ordering::Release);
}
```

On strongly-ordered platforms most accesses have release or acquire semantics, making release and acquire often totally free. This is not the case on weakly-ordered platforms.

Relaxed

Relaxed accesses are the absolute weakest. They can be freely re-ordered and provide no happens-before relationship. Still, relaxed operations are still atomic. That is, they don't count as data accesses and any read-modify-write operations done to them occur atomically. Relaxed operations are appropriate for things that you definitely want to happen, but don't particularly otherwise care about. For instance, incrementing a counter can be safely done by multiple threads using a relaxed `fetch_add` if you're not using the counter to synchronize any other accesses.

There's rarely a benefit in making an operation relaxed on strongly-ordered platforms, since they usually provide release-acquire semantics anyway. However relaxed operations can be cheaper on weakly-ordered platforms.

10

Implementing Vec

To bring everything together, we're going to write `std::Vec` from scratch. Because all the best tools for writing unsafe code are unstable, this project will only work on nightly (as of Rust 1.2.0). With the exception of the allocator API, much of the unstable code we'll use is expected to be stabilized in a similar form as it is today.

However we will generally try to avoid unstable code where possible. In particular we won't use any intrinsics that could make a code a little bit nicer or efficient because intrinsics are permanently unstable. Although many intrinsics *do* become stabilized elsewhere (`std::ptr` and `str::mem` consist of many intrinsics).

Ultimately this means our implementation may not take advantage of all possible optimizations, though it will be by no means *naive*. We will definitely get into the weeds over nitty-gritty details, even when the problem doesn't *really* merit it.

You wanted advanced. We're gonna go advanced.

Layout

First off, we need to come up with the struct layout. A Vec has three parts: a pointer to the allocation, the size of the allocation, and the number of elements that have been initialized.

Naively, this means we just want this design:

```
pub struct Vec<T> {
    ptr: *mut T,
    cap: usize,
    len: usize,
}
```

And indeed this would compile. Unfortunately, it would be incorrect. First, the compiler will give us too strict variance. So a `&Vec<&'static str>` couldn't be used where an `&Vec<&'a str>`

was expected. More importantly, it will give incorrect ownership information to the drop checker, as it will conservatively assume we don't own any values of type `T`. See the chapter on ownership and lifetimes (chapter 4, page 25) for all the details on variance and drop check.

As we saw in the ownership chapter, we should use `Unique<T>` in place of `*mut T` when we have a raw pointer to an allocation we own. Unique is unstable, so we'd like to not use it if possible, though.

As a recap, Unique is a wrapper around a raw pointer that declares that:

- We are variant over `T`
- We may own a value of type `T` (for drop check)
- We are Send/Sync if `T` is Send/Sync
- We deref to `*mut T` (so it largely acts like a `*mut` in our code)
- Our pointer is never null (so `Option<Vec<T>>` is null-pointer-optimized)

We can implement all of the above requirements except for the last one in stable Rust:

```
use std::marker::PhantomData;
use std::ops::Deref;
use std::mem;

struct Unique<T> {
    ptr: *const T,              // *const for variance
    _marker: PhantomData<T>,    // For the drop checker
}

// Deriving Send and Sync is safe because we are the Unique owners
// of this data. It's like Unique<T> is "just" T.
unsafe impl<T: Send> Send for Unique<T> {}
unsafe impl<T: Sync> Sync for Unique<T> {}

impl<T> Unique<T> {
    pub fn new(ptr: *mut T) -> Self {
        Unique { ptr: ptr, _marker: PhantomData }
    }
}

impl<T> Deref for Unique<T> {
    type Target = *mut T;
    fn deref(&self) -> &*mut T {
        // There's no way to cast the *const to a *mut
        // while also taking a reference. So we just
        // transmute it since it's all "just pointers".
        unsafe { mem::transmute(&self.ptr) }
    }
}
```

Unfortunately the mechanism for stating that your value is non-zero is unstable and unlikely to be stabilized soon. As such we're just going to take the hit and use std's Unique:

```
#![feature(unique)]

use std::ptr::{Unique, self};

pub struct Vec<T> {
    ptr: Unique<T>,
    cap: usize,
    len: usize,
}
```

If you don't care about the null-pointer optimization, then you can use the stable code. However we will be designing the rest of the code around enabling the optimization. In particular, `Unique::new` is unsafe to call, because putting `null` inside of it is Undefined Behaviour. Our stable Unique doesn't need `new` to be unsafe because it doesn't make any interesting guarantees about its contents.

Allocating

Using Unique throws a wrench in an important feature of Vec (and indeed all of the std collections): an empty Vec doesn't actually allocate at all. So if we can't allocate, but also can't put a null pointer in `ptr`, what do we do in `Vec::new`? Well, we just put some other garbage in there!

This is perfectly fine because we already have `cap == 0` as our sentinel for no allocation. We don't even need to handle it specially in almost any code because we usually need to check if `cap > len` or `len > 0` anyway. The traditional Rust value to put here is `0x01`. The standard library actually exposes this as `alloc::heap::EMPTY`. There are quite a few places where we'll want to use `heap::EMPTY` because there's no real allocation to talk about but `null` would make the compiler do bad things.

All of the `heap` API is totally unstable under the `heap_api` feature, though. We could trivially define `heap::EMPTY` ourselves, but we'll want the rest of the `heap` API anyway, so let's just get that dependency over with.

So:

```
#![feature(alloc, heap_api)]

use std::mem;

use alloc::heap::EMPTY;

impl<T> Vec<T> {
    fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "We're not ready to handle ZSTs");
        unsafe {
            // need to cast EMPTY to the actual ptr type we want, let
            // inference handle it.
            Vec { ptr: Unique::new(heap::EMPTY as *mut _), len: 0, cap: 0 }
```

```
        }
    }
}
```

I slipped in that assert there because zero-sized types will require some special handling throughout our code, and I want to defer the issue for now. Without this assert, some of our early drafts will do some Very Bad Things.

Next we need to figure out what to actually do when we *do* want space. For that, we'll need to use the rest of the heap APIs. These basically allow us to talk directly to Rust's allocator (jemalloc by default).

We'll also need a way to handle out-of-memory (OOM) conditions. The standard library calls the `abort` intrinsic, which just calls an illegal instruction to crash the whole program. The reason we abort and don't panic is because unwinding can cause allocations to happen, and that seems like a bad thing to do when your allocator just came back with "hey I don't have any more memory".

Of course, this is a bit silly since most platforms don't actually run out of memory in a conventional way. Your operating system will probably kill the application by another means if you legitimately start using up all the memory. The most likely way we'll trigger OOM is by just asking for ludicrous quantities of memory at once (e.g. half the theoretical address space). As such it's *probably* fine to panic and nothing bad will happen. Still, we're trying to be like the standard library as much as possible, so we'll just kill the whole program.

We said we don't want to use intrinsics, so doing exactly what `std` does is out. Instead, we'll call `std::process::exit` with some random number.

```
fn oom() {
    ::std::process::exit(-9999);
}
```

Okay, now we can write growing. Roughly, we want to have this logic:

```
if cap == 0:
    allocate()
    cap = 1
else:
    reallocate()
    cap *= 2
```

But Rust's only supported allocator API is so low level that we'll need to do a fair bit of extra work. We also need to guard against some special conditions that can occur with really large allocations or empty allocations.

In particular, `ptr::offset` will cause us a lot of trouble, because it has the semantics of LLVM's GEP inbounds instruction. If you're fortunate enough to not have dealt with this instruction, here's the basic story with GEP: alias analysis, alias analysis, alias analysis. It's super important to an optimizing compiler to be able to reason about data dependencies and aliasing.

As a simple example, consider the following fragment of code:

```
*x *= 7;
*y *= 3;
```

If the compiler can prove that `x` and `y` point to different locations in memory, the two operations can in theory be executed in parallel (by e.g. loading them into different registers and working on them independently). However the compiler can't do this in general because if x and y point to the same location in memory, the operations need to be done to the same value, and they can't just be merged afterwards.

When you use GEP inbounds, you are specifically telling LLVM that the offsets you're about to do are within the bounds of a single "allocated" entity. The ultimate payoff being that LLVM can assume that if two pointers are known to point to two disjoint objects, all the offsets of those pointers are *also* known to not alias (because you won't just end up in some random place in memory). LLVM is heavily optimized to work with GEP offsets, and inbounds offsets are the best of all, so it's important that we use them as much as possible.

So that's what GEP's about, how can it cause us trouble?

The first problem is that we index into arrays with unsigned integers, but GEP (and as a consequence `ptr::offset`) takes a signed integer. This means that half of the seemingly valid indices into an array will overflow GEP and actually go in the wrong direction! As such we must limit all allocations to `isize::MAX` elements. This actually means we only need to worry about byte-sized objects, because e.g. `> isize::MAX u16`s will truly exhaust all of the system's memory. However in order to avoid subtle corner cases where someone reinterprets some array of `< isize::MAX` objects as bytes, std limits all allocations to `isize::MAX` bytes.

On all 64-bit targets that Rust currently supports we're artificially limited to significantly less than all 64 bits of the address space (modern x64 platforms only expose 48-bit addressing), so we can rely on just running out of memory first. However on 32-bit targets, particularly those with extensions to use more of the address space (PAE x86 or x32), it's theoretically possible to successfully allocate more than `isize::MAX` bytes of memory.

However since this is a tutorial, we're not going to be particularly optimal here, and just unconditionally check, rather than use clever platform-specific `cfg`s.

The other corner-case we need to worry about is empty allocations. There will be two kinds of empty allocations we need to worry about: `cap = 0` for all T, and `cap > 0` for zero-sized types.

These cases are tricky because they come down to what LLVM means by "allocated". LLVM's notion of an allocation is significantly more abstract than how we usually use it. Because LLVM needs to work with different languages' semantics and custom allocators, it can't really intimately understand allocation. Instead, the main idea behind allocation is "doesn't overlap with other stuff". That is, heap allocations, stack allocations, and globals don't randomly overlap. Yep, it's about alias analysis. As such, Rust can technically play a bit fast an loose with the notion of an allocation as long as it's *consistent*.

Getting back to the empty allocation case, there are a couple of places where we want to offset by 0 as a consequence of generic code. The question is then: is it consistent to do so? For zero-sized types, we have concluded that it is indeed consistent to do a GEP inbounds offset by an arbitrary number of elements. This is a runtime no-op because every element takes up no space, and it's fine to pretend that there's infinite zero-sized types allocated at `0x01`. No allocator will ever allocate that address, because they won't allocate `0x00` and they generally allocate to

some minimal alignment higher than a byte. Also generally the whole first page of memory is protected from being allocated anyway (a whole 4k, on many platforms).

However what about for positive-sized types? That one's a bit trickier. In principle, you can argue that offsetting by 0 gives LLVM no information: either there's an element before the address or after it, but it can't know which. However we've chosen to conservatively assume that it may do bad things. As such we will guard against this case explicitly.

Phew

Ok with all the nonsense out of the way, let's actually allocate some memory:

```
fn grow(&mut self) {
    // this is all pretty delicate, so let's say it's all unsafe
    unsafe {
        // current API requires us to specify size and alignment manually.
        let align = mem::align_of::<T>();
        let elem_size = mem::size_of::<T>();

        let (new_cap, ptr) = if self.cap == 0 {
            let ptr = heap::allocate(elem_size, align);
            (1, ptr)
        } else {
            // as an invariant, we can assume that `self.cap < isize::MAX`,
            // so this doesn't need to be checked.
            let new_cap = self.cap * 2;
            // Similarly this can't overflow due to previously allocating this
            let old_num_bytes = self.cap * elem_size;

            // check that the new allocation doesn't exceed `isize::MAX` at all
            // regardless of the actual size of the capacity. This combines the
            // `new_cap <= isize::MAX` and `new_num_bytes <= usize::MAX` checks
            // we need to make. We lose the ability to allocate e.g. 2/3rds of
            // the address space with a single Vec of i16's on 32-bit though.
            // Alas, poor Yorick -- I knew him, Horatio.
            assert!(old_num_bytes <= (::std::isize::MAX as usize) / 2,
                    "capacity overflow");

            let new_num_bytes = old_num_bytes * 2;
            let ptr = heap::reallocate(*self.ptr as *mut _,
                                       old_num_bytes,
                                       new_num_bytes,
                                       align);
            (new_cap, ptr)
        };

        // If allocate or reallocate fail, we'll get `null` back
        if ptr.is_null() { oom(); }

        self.ptr = Unique::new(ptr as *mut _);
        self.cap = new_cap;
```

```
    }
}
```

Nothing particularly tricky here. Just computing sizes and alignments and doing some careful multiplication checks.

Push and Pop

Alright. We can initialize. We can allocate. Let's actually implement some functionality! Let's start with `push`. All it needs to do is check if we're full to grow, unconditionally write to the next index, and then increment our length.

To do the write we have to be careful not to evaluate the memory we want to write to. At worst, it's truly uninitialized memory from the allocator. At best it's the bits of some old value we popped off. Either way, we can't just index to the memory and dereference it, because that will evaluate the memory as a valid instance of T. Worse, `foo[idx] = x` will try to call `drop` on the old value of `foo[idx]`!

The correct way to do this is with `ptr::write`, which just blindly overwrites the target address with the bits of the value we provide. No evaluation involved.

For `push`, if the old len (before push was called) is 0, then we want to write to the 0th index. So we should offset by the old len.

```
pub fn push(&mut self, elem: T) {
    if self.len == self.cap { self.grow(); }

    unsafe {
        ptr::write(self.ptr.offset(self.len as isize), elem);
    }

    // Can't fail, we'll OOM first.
    self.len += 1;
}
```

Easy! How about `pop`? Although this time the index we want to access is initialized, Rust won't just let us dereference the location of memory to move the value out, because that would leave the memory uninitialized! For this we need `ptr::read`, which just copies out the bits from the target address and intrprets it as a value of type T. This will leave the memory at this address logically uninitialized, even though there is in fact a perfectly good instance of T there.

For `pop`, if the old len is 1, we want to read out of the 0th index. So we should offset by the new len.

```
pub fn pop(&mut self) -> Option<T> {
    if self.len == 0 {
        None
    } else {
```

```
        self.len -= 1;
        unsafe {
            Some(ptr::read(self.ptr.offset(self.len as isize)))
        }
    }
}
```

Deallocating

Next we should implement Drop so that we don't massively leak tons of resources. The easiest way is to just call `pop` until it yields None, and then deallocate our buffer. Note that calling `pop` is unneeded if `T: !Drop`. In theory we can ask Rust if `T needs_drop` and omit the calls to `pop`. However in practice LLVM is *really* good at removing simple side-effect free code like this, so I wouldn't bother unless you notice it's not being stripped (in this case it is).

We must not call `heap::deallocate` when `self.cap == 0`, as in this case we haven't actually allocated any memory.

```
impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            while let Some(_) = self.pop() { }

            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr, num_bytes, align);
            }
        }
    }
}
```

Deref

Alright! We've got a decent minimal stack implemented. We can push, we can pop, and we can clean up after ourselves. However there's a whole mess of functionality we'd reasonably want. In particular, we have a proper array, but none of the slice functionality. That's actually pretty easy to solve: we can implement `Deref<Target=[T]>`. This will magically make our Vec coerce to, and behave like, a slice in all sorts of conditions.

All we need is `slice::from_raw_parts`. It will correctly handle empty slices for us. Later once we set up zero-sized type support it will also Just Work for those too.

```
use std::ops::Deref;
```

```
impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe {
            ::std::slice::from_raw_parts(*self.ptr, self.len)
        }
    }
}
```

And let's do DerefMut too:

```
use std::ops::DerefMut;

impl<T> DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe {
            ::std::slice::from_raw_parts_mut(*self.ptr, self.len)
        }
    }
}
```

Now we have `len`, `first`, `last`, indexing, slicing, sorting, `iter`, `iter_mut`, and all other sorts of bells and whistles provided by slice. Sweet!

Insert and Remove

Something *not* provided by slice is `insert` and `remove`, so let's do those next.

Insert needs to shift all the elements at the target index to the right by one. To do this we need to use `ptr::copy`, which is our version of C's `memmove`. This copies some chunk of memory from one location to another, correctly handling the case where the source and destination overlap (which will definitely happen here).

If we insert at index `i`, we want to shift the `[i .. len]` to `[i+1 .. len+1]` using the old len.

```
pub fn insert(&mut self, index: usize, elem: T) {
    // Note: `<=` because it's valid to insert after everything
    // which would be equivalent to push.
    assert!(index <= self.len, "index out of bounds");
    if self.cap == self.len { self.grow(); }

    unsafe {
        if index < self.len {
            // ptr::copy(src, dest, len): "copy from source to dest len elems"
            ptr::copy(self.ptr.offset(index as isize),
                      self.ptr.offset(index as isize + 1),
                      len - index);
```

```
        }
        ptr::write(self.ptr.offset(index as isize), elem);
        self.len += 1;
    }
}
```

Remove behaves in the opposite manner. We need to shift all the elements from `[i+1 .. len + 1]` to `[i .. len]` using the *new* len.

```
pub fn remove(&mut self, index: usize) -> T {
    // Note: `<` because it's *not* valid to remove after everything
    assert!(index < self.len, "index out of bounds");
    unsafe {
        self.len -= 1;
        let result = ptr::read(self.ptr.offset(index as isize));
        ptr::copy(self.ptr.offset(index as isize + 1),
                  self.ptr.offset(index as isize),
                  len - index);
        result
    }
}
```

IntoIter

Let's move on to writing iterators. `iter` and `iter_mut` have already been written for us thanks to The Magic of Deref. However there's two interesting iterators that Vec provides that slices can't: `into_iter` and `drain`.

IntoIter consumes the Vec by-value, and can consequently yield its elements by-value. In order to enable this, IntoIter needs to take control of Vec's allocation.

IntoIter needs to be DoubleEnded as well, to enable reading from both ends. Reading from the back could just be implemented as calling `pop`, but reading from the front is harder. We could call `remove(0)` but that would be insanely expensive. Instead we're going to just use ptr::read to copy values out of either end of the Vec without mutating the buffer at all.

To do this we're going to use a very common C idiom for array iteration. We'll make two pointers; one that points to the start of the array, and one that points to one-element past the end. When we want an element from one end, we'll read out the value pointed to at that end and move the pointer over by one. When the two pointers are equal, we know we're done.

Note that the order of read and offset are reversed for `next` and `next_back` For `next_back` the pointer is always after the element it wants to read next, while for `next` the pointer is always at the element it wants to read next. To see why this is, consider the case where every element but one has been yielded.

The array looks like this:

```
         S  E
[X, X, X, O, X, X, X]
```

If E pointed directly at the element it wanted to yield next, it would be indistinguishable from the case where there are no more elements to yield.

Although we don't actually care about it during iteration, we also need to hold onto the Vec's allocation information in order to free it once IntoIter is dropped.

So we're going to use the following struct:

```
struct IntoIter<T> {
    buf: Unique<T>,
    cap: usize,
    start: *const T,
    end: *const T,
}
```

And this is what we end up with for initialization:

```
impl<T> Vec<T> {
    fn into_iter(self) -> IntoIter<T> {
        // Can't destructure Vec since it's Drop
        let ptr = self.ptr;
        let cap = self.cap;
        let len = self.len;

        // Make sure not to drop Vec since that will free the buffer
        mem::forget(self);

        unsafe {
            IntoIter {
                buf: ptr,
                cap: cap,
                start: *ptr,
                end: if cap == 0 {
                    // can't offset off this pointer, it's not allocated!
                    *ptr
                } else {
                    ptr.offset(len as isize)
                }
            }
        }
    }
}
```

Here's iterating forward:

```
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
```

```
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = self.start.offset(1);
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let len = (self.end as usize - self.start as usize)
                  / mem::size_of::<T>();
        (len, Some(len))
    }
}
```

And here's iterating backwards.

```
impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = self.end.offset(-1);
                Some(ptr::read(self.end))
            }
        }
    }
}
```

Because IntoIter takes ownership of its allocation, it needs to implement Drop to free it. However it also wants to implement Drop to drop any elements it contains that weren't yielded.

```
impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            // drop any remaining elements
            for _ in &mut *self {}

            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.buf as *mut _, num_bytes, align);
            }
        }
```

```
    }
}
```

RawVec

We've actually reached an interesting situation here: we've duplicated the logic for specifying a buffer and freeing its memory in Vec and IntoIter. Now that we've implemented it and identified *actual* logic duplication, this is a good time to perform some logic compression.

We're going to abstract out the `(ptr, cap)` pair and give them the logic for allocating, growing, and freeing:

```
struct RawVec<T> {
    ptr: Unique<T>,
    cap: usize,
}

impl<T> RawVec<T> {
    fn new() -> Self {
        assert!(mem::size_of::<T>() != 0, "TODO: implement ZST support");
        unsafe {
            RawVec { ptr: Unique::new(heap::EMPTY as *mut T), cap: 0 }
        }
    }

    // unchanged from Vec
    fn grow(&mut self) {
        unsafe {
            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(*self.ptr as *mut _,
                                           self.cap * elem_size,
                                           new_cap * elem_size,
                                           align);
                (new_cap, ptr)
            };

            // If allocate or reallocate fail, we'll get `null` back
            if ptr.is_null() { oom() }

            self.ptr = Unique::new(ptr as *mut _);
```

```
            self.cap = new_cap;
        }
    }
}


impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        if self.cap != 0 {
            let align = mem::align_of::<T>();
            let elem_size = mem::size_of::<T>();
            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
            }
        }
    }
}
```

And change Vec as follows:

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl<T> Vec<T> {
    fn ptr(&self) -> *mut T { *self.buf.ptr }

    fn cap(&self) -> usize { self.buf.cap }

    pub fn new() -> Self {
        Vec { buf: RawVec::new(), len: 0 }
    }

    // push/pop/insert/remove largely unchanged:
    // * `self.ptr -> self.ptr()`
    // * `self.cap -> self.cap()`
    // * `self.grow -> self.buf.grow()`
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() {}
        // deallocation is handled by RawVec
    }
}
```

And finally we can really simplify IntoIter:

```
struct IntoIter<T> {
    _buf: RawVec<T>, // we don't actually care about this. Just need it to live.
    start: *const T,
    end: *const T,
}

// next and next_back literally unchanged since they never referred to the buf

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        // only need to ensure all our elements are read;
        // buffer will clean itself up afterwards.
        for _ in &mut *self {}
    }
}

impl<T> Vec<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            // need to use ptr::read to unsafely move the buf out since it's
            // not Copy, and Vec implements Drop (so we can't destructure it).
            let buf = ptr::read(&self.buf);
            let len = self.len;
            mem::forget(self);

            IntoIter {
                start: *buf.ptr,
                end: buf.ptr.offset(len as isize),
                _buf: buf,
            }
        }
    }
}
```

Much better.

Drain

Let's move on to Drain. Drain is largely the same as IntoIter, except that instead of consuming the Vec, it borrows the Vec and leaves its allocation untouched. For now we'll only implement the "basic" full-range version.

```
use std::marker::PhantomData;

struct Drain<'a, T: 'a> {
    // Need to bound the lifetime here, so we do it with `&'a mut Vec<T>`
    // because that's semantically what we contain. We're "just" calling
```

```
    // `pop()` and `remove(0)`.
    vec: PhantomData<&'a mut Vec<T>>
    start: *const T,
    end: *const T,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
```

– wait, this is seeming familiar. Let's do some more compression. Both IntoIter and Drain have the exact same structure, let's just factor it out.

```
struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    // unsafe to construct because it has no associated lifetimes.
    // This is necessary to store a RawValIter in the same struct as
    // its actual allocation. OK since it's a private implementation
    // detail.
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if slice.len() == 0 {
                // if `len = 0`, then this is not actually allocated memory.
                // Need to avoid offsetting because that will give wrong
                // information to LLVM via GEP.
                slice.as_ptr()
            } else {
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}

// Iterator and DoubleEndedIterator impls identical to IntoIter.
```

And IntoIter becomes the following:

```
pub struct IntoIter<T> {
    _buf: RawVec<T>, // we don't actually care about this. Just need it to live.
    iter: RawValIter<T>,
}
```

```
impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut self.iter {}
    }
}

impl<T> Vec<T> {
    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            let buf = ptr::read(&self.buf);
            mem::forget(self);

            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
    }
}
```

Note that I've left a few quirks in this design to make upgrading Drain to work with arbitrary subranges a bit easier. In particular we *could* have RawValIter drain itself on drop, but that won't work right for a more complex Drain. We also take a slice to simplify Drain initialization.

Alright, now Drain is really easy:

```
use std::marker::PhantomData;

pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
```

```
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        for _ in &mut self.iter {}
    }
}

impl<T> Vec<T> {
    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // this is a mem::forget safety thing. If Drain is forgotten, we just
            // leak the whole Vec's contents. Also we need to do this *eventually*
            // anyway, so why not do it now?
            self.len = 0;

            Drain {
                iter: iter,
                vec: PhantomData,
            }
        }
    }
}
```

For more details on the `mem::forget` problem, see the section on leaks (section 7, page 68).

Handling Zero-Sized Types

It's time. We're going to fight the spectre that is zero-sized types. Safe Rust *never* needs to care about this, but Vec is very intensive on raw pointers and raw allocations, which are exactly the two things that care about zero-sized types. We need to be careful of two things:

- The raw allocator API has undefined behaviour if you pass in 0 for an allocation size.
- raw pointer offsets are no-ops for zero-sized types, which will break our C-style pointer iterator.

Thankfully we abstracted out pointer-iterators and allocating handling into RawValIter and RawVec respectively. How mysteriously convenient.

Allocating Zero-Sized Types

So if the allocator API doesn't support zero-sized allocations, what on earth do we store as our allocation? Why, `heap::EMPTY` of course! Almost every operation with a ZST is a no-op since ZSTs have exactly one value, and therefore no state needs to be considered to store or load them. This actually extends to `ptr::read` and `ptr::write`: they won't actually look at the pointer at all. As such we never need to change the pointer.

Note however that our previous reliance on running out of memory before overflow is no longer valid with zero-sized types. We must explicitly guard against capacity overflow for zero-sized types.

Due to our current architecture, all this means is writing 3 guards, one in each method of RawVec.

```
impl<T> RawVec<T> {
    fn new() -> Self {
        unsafe {
            // !0 is usize::MAX. This branch should be stripped at compile time.
            let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

            // heap::EMPTY doubles as "unallocated" and "zero-sized allocation"
            RawVec { ptr: Unique::new(heap::EMPTY as *mut T), cap: cap }
        }
    }

    fn grow(&mut self) {
        unsafe {
            let elem_size = mem::size_of::<T>();

            // since we set the capacity to usize::MAX when elem_size is
            // 0, getting to here necessarily means the Vec is overfull.
            assert!(elem_size != 0, "capacity overflow");

            let align = mem::align_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(*self.ptr as *mut _,
                                           self.cap * elem_size,
                                           new_cap * elem_size,
                                           align);
                (new_cap, ptr)
            };

            // If allocate or reallocate fail, we'll get `null` back
            if ptr.is_null() { oom() }
```

```
            self.ptr = Unique::new(ptr as *mut _);
            self.cap = new_cap;
        }
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        let elem_size = mem::size_of::<T>();

        // don't free zero-sized allocations, as they were never allocated.
        if self.cap != 0 && elem_size != 0 {
            let align = mem::align_of::<T>();

            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
            }
        }
    }
}
```

That's it. We support pushing and popping zero-sized types now. Our iterators (that aren't provided by slice Deref) are still busted, though.

Iterating Zero-Sized Types

Zero-sized offsets are no-ops. This means that our current design will always initialize `start` and `end` as the same value, and our iterators will yield nothing. The current solution to this is to cast the pointers to integers, increment, and then cast them back:

```
impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}
```

Now we have a different bug. Instead of our iterators not running at all, our iterators now run *forever*. We need to do the same trick in our iterator impls. Also, our size_hint computation

code will divide by 0 for ZSTs. Since we'll basically be treating the two pointers as if they point to bytes, we'll just map size 0 to divide by 1.

```
impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = if mem::size_of::<T>() == 0 {
                    (self.start as usize + 1) as *const _
                } else {
                    self.start.offset(1);
                }
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
        (len, Some(len))
    }
}

impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = if mem::size_of::<T>() == 0 {
                    (self.end as usize - 1) as *const _
                } else {
                    self.end.offset(-1);
                }
                Some(ptr::read(self.end))
            }
        }
    }
}
```

And that's it. Iteration works!

Final Code

```
#![feature(unique)]
#![feature(alloc, heap_api)]

extern crate alloc;

use std::ptr::{Unique, self};
use std::mem;
use std::ops::{Deref, DerefMut};
use std::marker::PhantomData;

use alloc::heap;

struct RawVec<T> {
    ptr: Unique<T>,
    cap: usize,
}

impl<T> RawVec<T> {
    fn new() -> Self {
        unsafe {
            // !0 is usize::MAX. This branch should be stripped at compile time.
            let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

            // heap::EMPTY doubles as "unallocated" and "zero-sized allocation"
            RawVec { ptr: Unique::new(heap::EMPTY as *mut T), cap: cap }
        }
    }

    fn grow(&mut self) {
        unsafe {
            let elem_size = mem::size_of::<T>();

            // since we set the capacity to usize::MAX when elem_size is
            // 0, getting to here necessarily means the Vec is overfull.
            assert!(elem_size != 0, "capacity overflow");

            let align = mem::align_of::<T>();

            let (new_cap, ptr) = if self.cap == 0 {
                let ptr = heap::allocate(elem_size, align);
                (1, ptr)
            } else {
                let new_cap = 2 * self.cap;
                let ptr = heap::reallocate(*self.ptr as *mut _,
                                           self.cap * elem_size,
                                           new_cap * elem_size,
```

```
                                            align);
                (new_cap, ptr)
            };

            // If allocate or reallocate fail, we'll get `null` back
            if ptr.is_null() { oom() }

            self.ptr = Unique::new(ptr as *mut _);
            self.cap = new_cap;
        }
    }
}

impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
        let elem_size = mem::size_of::<T>();
        if self.cap != 0 && elem_size != 0 {
            let align = mem::align_of::<T>();

            let num_bytes = elem_size * self.cap;
            unsafe {
                heap::deallocate(*self.ptr as *mut _, num_bytes, align);
            }
        }
    }
}




pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}

impl<T> Vec<T> {
    fn ptr(&self) -> *mut T { *self.buf.ptr }

    fn cap(&self) -> usize { self.buf.cap }

    pub fn new() -> Self {
        Vec { buf: RawVec::new(), len: 0 }
    }
    pub fn push(&mut self, elem: T) {
        if self.len == self.cap() { self.buf.grow(); }

        unsafe {
```

```
            ptr::write(self.ptr().offset(self.len as isize), elem);
        }

        // Can't fail, we'll OOM first.
        self.len += 1;
    }

    pub fn pop(&mut self) -> Option<T> {
        if self.len == 0 {
            None
        } else {
            self.len -= 1;
            unsafe {
                Some(ptr::read(self.ptr().offset(self.len as isize)))
            }
        }
    }

    pub fn insert(&mut self, index: usize, elem: T) {
        assert!(index <= self.len, "index out of bounds");
        if self.cap() == self.len { self.buf.grow(); }

        unsafe {
            if index < self.len {
                ptr::copy(self.ptr().offset(index as isize),
                          self.ptr().offset(index as isize + 1),
                          self.len - index);
            }
            ptr::write(self.ptr().offset(index as isize), elem);
            self.len += 1;
        }
    }

    pub fn remove(&mut self, index: usize) -> T {
        assert!(index < self.len, "index out of bounds");
        unsafe {
            self.len -= 1;
            let result = ptr::read(self.ptr().offset(index as isize));
            ptr::copy(self.ptr().offset(index as isize + 1),
                      self.ptr().offset(index as isize),
                      self.len - index);
            result
        }
    }

    pub fn into_iter(self) -> IntoIter<T> {
        unsafe {
            let iter = RawValIter::new(&self);
```

```
            let buf = ptr::read(&self.buf);
            mem::forget(self);

            IntoIter {
                iter: iter,
                _buf: buf,
            }
        }
    }

    pub fn drain(&mut self) -> Drain<T> {
        unsafe {
            let iter = RawValIter::new(&self);

            // this is a mem::forget safety thing. If Drain is forgotten, we just
            // leak the whole Vec's contents. Also we need to do this *eventually*
            // anyway, so why not do it now?
            self.len = 0;

            Drain {
                iter: iter,
                vec: PhantomData,
            }
        }
    }
}

impl<T> Drop for Vec<T> {
    fn drop(&mut self) {
        while let Some(_) = self.pop() {}
        // allocation is handled by RawVec
    }
}

impl<T> Deref for Vec<T> {
    type Target = [T];
    fn deref(&self) -> &[T] {
        unsafe {
            ::std::slice::from_raw_parts(self.ptr(), self.len)
        }
    }
}

impl<T> DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe {
            ::std::slice::from_raw_parts_mut(self.ptr(), self.len)
        }
```

```
    }
}




struct RawValIter<T> {
    start: *const T,
    end: *const T,
}

impl<T> RawValIter<T> {
    unsafe fn new(slice: &[T]) -> Self {
        RawValIter {
            start: slice.as_ptr(),
            end: if mem::size_of::<T>() == 0 {
                ((slice.as_ptr() as usize) + slice.len()) as *const _
            } else if slice.len() == 0 {
                slice.as_ptr()
            } else {
                slice.as_ptr().offset(slice.len() as isize)
            }
        }
    }
}

impl<T> Iterator for RawValIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                let result = ptr::read(self.start);
                self.start = self.start.offset(1);
                Some(result)
            }
        }
    }

    fn size_hint(&self) -> (usize, Option<usize>) {
        let elem_size = mem::size_of::<T>();
        let len = (self.end as usize - self.start as usize)
                  / if elem_size == 0 { 1 } else { elem_size };
        (len, Some(len))
    }
}
```

```
impl<T> DoubleEndedIterator for RawValIter<T> {
    fn next_back(&mut self) -> Option<T> {
        if self.start == self.end {
            None
        } else {
            unsafe {
                self.end = self.end.offset(-1);
                Some(ptr::read(self.end))
            }
        }
    }
}




pub struct IntoIter<T> {
    _buf: RawVec<T>, // we don't actually care about this. Just need it to live.
    iter: RawValIter<T>,
}

impl<T> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next() }
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<T> DoubleEndedIterator for IntoIter<T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<T> Drop for IntoIter<T> {
    fn drop(&mut self) {
        for _ in &mut *self {}
    }
}




pub struct Drain<'a, T: 'a> {
    vec: PhantomData<&'a mut Vec<T>>,
    iter: RawValIter<T>,
}

impl<'a, T> Iterator for Drain<'a, T> {
    type Item = T;
    fn next(&mut self) -> Option<T> { self.iter.next_back() }
```

```
    fn size_hint(&self) -> (usize, Option<usize>) { self.iter.size_hint() }
}

impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
    fn next_back(&mut self) -> Option<T> { self.iter.next_back() }
}

impl<'a, T> Drop for Drain<'a, T> {
    fn drop(&mut self) {
        // pre-drain the iter
        for _ in &mut self.iter {}
    }
}

/// Abort the process, we're out of memory!
///
/// In practice this is probably dead code on most OSes
fn oom() {
    ::std::process::exit(-9999);
}
```

11

Implementing Arc and Mutex

Knowing the theory is all fine and good, but the *best* way to understand something is to use it. To better understand atomics and interior mutability, we'll be implementing versions of the standard library's Arc and Mutex types.

TODO: ALL OF THIS OMG